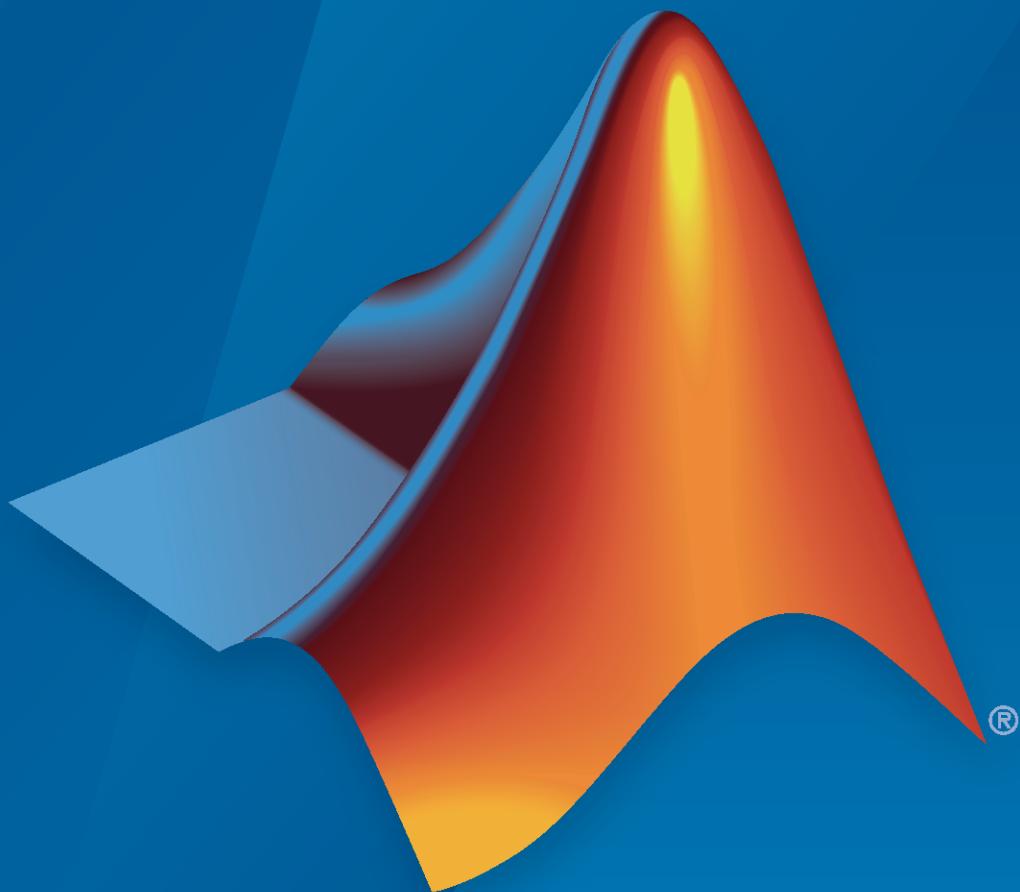


MATLAB®

Programming Fundamentals



MATLAB®

R2021a

 MathWorks®

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.
1 Apple Hill Drive
Natick, MA 01760-2098

MATLAB Programming Fundamentals

© COPYRIGHT 1984–2021 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)
March 2019	Online only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online only	Revised for MATLAB 9.10 (Release 2021a)

Language

Syntax Basics

1

Continue Long Statements on Multiple Lines	1-2
Ignore Function Outputs	1-3
Variable Names	1-4
Valid Names	1-4
Conflicts with Function Names	1-4
Case and Space Sensitivity	1-5
Choose Command Syntax or Function Syntax	1-6
Command Syntax and Function Syntax	1-6
Avoid Common Syntax Mistakes	1-7
How MATLAB Recognizes Command Syntax	1-7
Resolve Error: Undefined Function or Variable	1-9
Issue	1-9
Possible Solutions	1-9

Program Components

2

MATLAB Operators and Special Characters	2-2
Arithmetic Operators	2-2
Relational Operators	2-2
Logical Operators	2-2
Special Characters	2-3
String and Character Formatting	2-16
Array vs. Matrix Operations	2-20
Introduction	2-20
Array Operations	2-20
Matrix Operations	2-22
Compatible Array Sizes for Basic Operations	2-25
Inputs with Compatible Sizes	2-25

Inputs with Incompatible Sizes	2-27
Examples	2-27
Array Comparison with Relational Operators	2-29
Array Comparison	2-29
Logic Statements	2-31
Operator Precedence	2-32
Precedence of AND and OR Operators	2-32
Overriding Default Precedence	2-32
Average Similar Data Points Using a Tolerance	2-34
Group Scattered Data Using a Tolerance	2-36
Bit-Wise Operations	2-38
Perform Cyclic Redundancy Check	2-44
Conditional Statements	2-47
Loop Control Statements	2-49
Regular Expressions	2-51
What Is a Regular Expression?	2-51
Steps for Building Expressions	2-52
Operators and Characters	2-55
Lookahead Assertions in Regular Expressions	2-63
Lookahead Assertions	2-63
Overlapping Matches	2-63
Logical AND Conditions	2-64
Tokens in Regular Expressions	2-66
Introduction	2-66
Multiple Tokens	2-68
Unmatched Tokens	2-69
Tokens in Replacement Text	2-69
Named Capture	2-70
Dynamic Regular Expressions	2-72
Introduction	2-72
Dynamic Match Expressions — <code>(??expr)</code>	2-73
Commands That Modify the Match Expression — <code>(??@cmd)</code>	2-73
Commands That Serve a Functional Purpose — <code>(@cmd)</code>	2-74
Commands in Replacement Expressions — <code>\$(cmd)</code>	2-76
Comma-Separated Lists	2-79
What Is a Comma-Separated List?	2-79
Generating a Comma-Separated List	2-79
Assigning Output from a Comma-Separated List	2-81
Assigning to a Comma-Separated List	2-81
How to Use the Comma-Separated Lists	2-82
Fast Fourier Transform Example	2-84

Alternatives to the eval Function	2-86
Why Avoid the eval Function?	2-86
Variables with Sequential Names	2-86
Files with Sequential Names	2-87
Function Names in Variables	2-87
Field Names in Variables	2-88
Error Handling	2-88

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	4-2
Arithmetic Operations on Integer Classes	4-4
Largest and Smallest Values for Integer Classes	4-4
Floating-Point Numbers	4-6
Double-Precision Floating Point	4-6
Single-Precision Floating Point	4-6
Creating Floating-Point Data	4-6
Arithmetic Operations on Floating-Point Numbers	4-8
Largest and Smallest Values for Floating-Point Classes	4-9
Accuracy of Floating-Point Data	4-10
Avoiding Common Problems with Floating-Point Arithmetic	4-11
Create Complex Numbers	4-13
Infinity and NaN	4-14
Infinity	4-14
NaN	4-14
Identifying Numeric Classes	4-16
Display Format for Numeric Values	4-17
Integer Arithmetic	4-19
Single Precision Math	4-26

5

Find Array Elements That Meet a Condition	5-2
Reduce Logical Arrays to Single Value	5-6

Characters and Strings**6**

Text in String and Character Arrays	6-2
Create String Arrays	6-5
Cell Arrays of Character Vectors	6-12
Create Cell Array of Character Vectors	6-12
Access Character Vectors in Cell Array	6-12
Convert Cell Arrays to String Arrays	6-13
Analyze Text Data with String Arrays	6-15
Test for Empty Strings and Missing Values	6-20
Formatting Text	6-24
Fields of the Formatting Operator	6-24
Setting Field Width and Precision	6-28
Restrictions on Using Identifiers	6-30
Compare Text	6-32
Search and Replace Text	6-37
Build Pattern Expressions	6-40
Convert Numeric Values to Text	6-45
Convert Text to Numeric Values	6-49
Unicode and ASCII Values	6-53
Hexadecimal and Binary Values	6-55
Frequently Asked Questions About String Arrays	6-59
Why Does Using Command Form With Strings Return An Error? ..	6-59
Why Do Strings in Cell Arrays Return an Error?	6-60
Why Does <code>length()</code> of String Return 1?	6-60
Why Does <code>isempty("")</code> Return 0?	6-61
Why Does Appending Strings Using Square Brackets Return Multiple Strings?	6-62

Update Your Code to Accept Strings	6-64
What Are String Arrays?	6-64
Recommended Approaches for String Adoption in Old APIs	6-64
How to Adopt String Arrays in Old APIs	6-66
Recommended Approaches for String Adoption in New Code	6-66
How to Maintain Compatibility in New Code	6-67
How to Manually Convert Input Arguments	6-68
How to Check Argument Data Types	6-68
Terminology for Character and String Arrays	6-70
Function Summary	6-72

Dates and Time

7

Represent Dates and Times in MATLAB	7-2
Specify Time Zones	7-5
Convert Date and Time to Julian Date or POSIX Time	7-7
Set Date and Time Display Format	7-10
Formats for Individual Date and Duration Arrays	7-10
datetime Display Format	7-10
duration Display Format	7-11
calendarDuration Display Format	7-11
Default datetime Format	7-12
Generate Sequence of Dates and Time	7-14
Sequence of Datetime or Duration Values Between Endpoints with Step Size	7-14
Add Duration or Calendar Duration to Create Sequence of Dates	7-16
Specify Length and Endpoints of Date or Duration Sequence	7-17
Sequence of Datetime Values Using Calendar Rules	7-17
Share Code and Data Across Locales	7-20
Write Locale-Independent Date and Time Code	7-20
Write Dates in Other Languages	7-21
Read Dates in Other Languages	7-21
Extract or Assign Date and Time Components ofDatetime Array	7-23
Combine Date and Time from Separate Variables	7-26
Date and Time Arithmetic	7-28
Compare Dates and Time	7-33
Plot Dates and Durations	7-36
Line Plot with Dates	7-36
Line Plot with Durations	7-37
Scatter Plot with Dates and Durations	7-39

Plots that Support Dates and Durations	7-40
Core Functions Supporting Date and Time Arrays	7-41
Convert Between Datetime Arrays, Numbers, and Text	7-42
Overview	7-42
Convert Between Datetime and Character Vectors	7-42
Convert Between Datetime and String Arrays	7-44
Convert Between Datetime and Date Vectors	7-44
Convert Serial Date Numbers to Datetime	7-45
Convert Datetime Arrays to Numeric Values	7-45
Carryover in Date Vectors and Strings	7-47
Converting Date Vector Returns Unexpected Output	7-48

Categorical Arrays

8

Create Categorical Arrays	8-2
Convert Text in Table Variables to Categorical	8-6
Plot Categorical Data	8-10
Compare Categorical Array Elements	8-16
Combine Categorical Arrays	8-19
Combine Categorical Arrays Using Multiplication	8-22
Access Data Using Categorical Arrays	8-24
Select Data By Category	8-24
Common Ways to Access Data Using Categorical Arrays	8-24
Work with Protected Categorical Arrays	8-30
Advantages of Using Categorical Arrays	8-34
Natural Representation of Categorical Data	8-34
Mathematical Ordering for Character Vectors	8-34
Reduce Memory Requirements	8-34
Ordinal Categorical Arrays	8-36
Order of Categories	8-36
How to Create Ordinal Categorical Arrays	8-36
Working with Ordinal Categorical Arrays	8-38
Core Functions Supporting Categorical Arrays	8-39

Create and Work with Tables	9-2
Add and Delete Table Rows	9-11
Add, Delete, and Rearrange Table Variables	9-14
Clean Messy and Missing Data in Tables	9-21
Modify Units, Descriptions, and Table Variable Names	9-26
Add Custom Properties to Tables and Timetables	9-29
Access Data in Tables	9-34
Summary of Table Indexing Syntaxes	9-34
Tables Containing Specified Rows and Variables	9-37
Extract Data Using Dot Notation and Logical Values	9-40
Dot Notation with Any Variable Name or Expression	9-42
Extract Data from Specified Rows and Variables	9-44
Calculations on Tables	9-47
Split Data into Groups and Calculate Statistics	9-51
Split Table Data Variables and Apply Functions	9-54
Advantages of Using Tables	9-58
Grouping Variables To Split Data	9-63
Grouping Variables	9-63
Group Definition	9-63
The Split-Apply-Combine Workflow	9-64
Missing Group Values	9-64
Changes to DimensionNames Property in R2016b	9-66

Create Timetables	10-2
Resample and Aggregate Data in Timetable	10-5
Combine Timetables and Synchronize Their Data	10-8
Retime and Synchronize Timetable Variables Using Different Methods	10-14

Select Times in Timetable	10-19
Clean Timetable with Missing, Duplicate, or Nonuniform Times	10-27
Using Row Labels in Table and Timetable Operations	10-36
Loma Prieta Earthquake Analysis	10-41
Preprocess and Explore Time-stamped Data Using timetable	10-51

Structures

11

Structure Arrays	11-2
Create Scalar Structure	11-2
Access Values in Fields	11-2
Index into Nonscalar Structure Array	11-4
Concatenate Structures	11-8
Generate Field Names from Variables	11-10
Access Data in Nested Structures	11-11
Access Elements of a Nonscalar Structure Array	11-13
Ways to Organize Data in Structure Arrays	11-15
Plane Organization	11-15
Element-by-Element Organization	11-16
Memory Requirements for Structure Array	11-18

Cell Arrays

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	12-9
Combine Cell Arrays	12-10

Pass Contents of Cell Arrays to Functions	12-11
Preallocate Memory for Cell Array	12-15
Cell vs. Structure Arrays	12-16
Multilevel Indexing to Access Parts of Cells	12-20

Function Handles

13

Create Function Handle	13-2
What Is a Function Handle?	13-2
Creating Function Handles	13-2
Anonymous Functions	13-3
Arrays of Function Handles	13-4
Saving and Loading Function Handles	13-4
Pass Function to Another Function	13-5
Call Local Functions Using Function Handles	13-6
Compare Function Handles	13-8

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	14-4
Create Map Object	14-6
Construct Empty Map Object	14-6
Construct Initialized Map Object	14-6
Combine Map Objects	14-7
Examine Contents of Map	14-8
Read and Write Using Key Index	14-9
Read From Map	14-9
Add Key/Value Pairs	14-10
Build Map with Concatenation	14-10
Modify Keys and Values in Map	14-13
Remove Keys and Values from Map	14-13
Modify Values	14-13
Modify Keys	14-14

Modify Copy of Map	14-14
Map to Different Value Types	14-15
Map to Structure Array	14-15
Map to Cell Array	14-16

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	15-8
Combining Logical and Double Types	15-8

Using Objects

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	16-5

Scripts and Functions**Scripts**

Create Scripts	18-2
Add Comments to Programs	18-3
Code Sections	18-5
Divide Your File into Code Sections	18-5
Evaluate Code Sections	18-5
Navigate Among Code Sections in a File	18-6
Example of Evaluating Code Sections	18-7
Change the Appearance of Code Sections	18-9
Use Code Sections with Control Statements and Functions	18-9
Scripts vs. Functions	18-12
Add Functions to Scripts	18-14
Create a Script with Local Functions	18-14
Run Scripts with Local Functions	18-14
Restrictions for Local Functions and Variables	18-15
Access Help for Local Functions	18-15

Live Scripts and Functions

What Is a Live Script or Function?	19-2
Differences with Plain Code Scripts and Functions	19-3
Requirements	19-4
Unsupported Features	19-5
Create Live Scripts in the Live Editor	19-6
Create Live Script	19-6
Add Code	19-6
Run Code	19-8
Display Output	19-8
Change View	19-9
Format Text	19-10
Save Live Scripts as Plain Code	19-11

Run Sections in Live Scripts	19-13
Divide Your File Into Sections	19-13
Evaluate Sections	19-13
Debug Code in the Live Editor	19-15
Show Output	19-15
Debug Using Run to Here	19-16
View Variable Value While Debugging	19-18
Pause a Running File	19-18
End Debugging Session	19-18
Step Into Functions	19-19
Add Breakpoints and Run	19-19
Modify Figures in Live Scripts	19-23
Explore Data	19-23
Update Code with Figure Changes	19-25
Add Formatting and Annotations	19-25
Add and Modify Multiple Subplots	19-27
Save and Print Figure	19-31
Format Files in the Live Editor	19-32
Change Fonts	19-34
Autoformatting	19-35
Insert Equations into the Live Editor	19-38
Insert Equation Interactively	19-38
Insert LaTeX Equation	19-40
Add Interactive Controls to a Live Script	19-47
Insert Controls	19-47
Modify Control Execution	19-49
Modify Control Labels	19-49
Create Live Script with Multiple Interactive Controls	19-50
Share Live Script	19-51
Add Interactive Tasks to a Live Script	19-53
What Are Live Editor Tasks?	19-53
Insert Tasks	19-53
Run Tasks and Surrounding Code	19-56
Modify Output Argument Name	19-57
View and Edit Generated Code	19-57
Create Live Functions	19-59
Create Live Function	19-59
Add Code	19-59
Add Help	19-60
Run Live Function	19-60
Save Live Functions as Plain Code	19-61
Add Help for Live Functions	19-62
Share Live Scripts and Functions	19-66
Hide Code Before Sharing	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	19-70
Accelerate Exploratory Programming Using the Live Editor	19-75
Create an Interactive Narrative with the Live Editor	19-80
Create Interactive Course Materials Using the Live Editor	19-88
Create Examples Using the Live Editor	19-94
Create an Interactive Form Using the Live Editor	19-95
Create a Real-time Dashboard Using the Live Editor	19-98
Acknowledgments	19-99

20

Function Basics

Create Functions in Files	20-2
Syntax for Function Definition	20-2
Contents of Functions and Files	20-3
End Statements	20-4
Add Help for Your Program	20-5
Configure the Run Button for Functions	20-7
Base and Function Workspaces	20-9
Share Data Between Workspaces	20-10
Introduction	20-10
Best Practice: Passing Arguments	20-10
Nested Functions	20-10
Persistent Variables	20-11
Global Variables	20-12
Evaluating in Another Workspace	20-12
Check Variable Scope in Editor	20-14
Use Automatic Function and Variable Highlighting	20-14
Example of Using Automatic Function and Variable Highlighting	20-14
Types of Functions	20-17
Local and Nested Functions in a File	20-17
Private Functions in a Subfolder	20-18
Anonymous Functions Without a File	20-18

Anonymous Functions	20-20
What Are Anonymous Functions?	20-20
Variables in the Expression	20-21
Multiple Anonymous Functions	20-21
Functions with No Inputs	20-22
Functions with Multiple Inputs or Outputs	20-22
Arrays of Anonymous Functions	20-23
Local Functions	20-25
Nested Functions	20-27
What Are Nested Functions?	20-27
Requirements for Nested Functions	20-27
Sharing Variables Between Parent and Nested Functions	20-28
Using Handles to Store Function Parameters	20-29
Visibility of Nested Functions	20-31
Resolve Error: Attempt to Add Variable to a Static Workspace.	20-33
Issue	20-33
Possible Solutions	20-33
Private Functions	20-36
Function Precedence Order	20-37
Change in Rules For Function Precedence Order	20-38
Update Code for R2019b Changes to Function Precedence Order	20-40
Identifiers cannot be used for two purposes inside a function	20-40
Identifiers without explicit declarations might not be treated as variables	20-40
Variables cannot be implicitly shared between parent and nested functions	20-41
Change in precedence of wildcard-based imports	20-42
Fully qualified import functions cannot have the same name as nested functions	20-42
Fully qualified imports shadow outer scope definitions of the same name	20-43
Error handling when import not found	20-43
Nested functions inherit import statements from parent functions	20-44
Change in precedence of compound name resolution	20-44
Anonymous functions can include resolved and unresolved identifiers	20-45
Indexing into Function Call Results	20-46
Example	20-46
Supported Syntaxes	20-46

21

Find Number of Function Arguments	21-2
Support Variable Number of Inputs	21-4
Support Variable Number of Outputs	21-5
Validate Number of Function Arguments	21-6
Checking Number of Arguments in Nested Functions	21-8
Ignore Inputs in Function Definitions	21-10
Check Function Inputs with validateattributes	21-11
Parse Function Inputs	21-13
Input Parser Validation Functions	21-17

22

Debug a MATLAB Program	22-2
Set Breakpoint	22-2
Run File	22-3
Pause a Running File	22-3
Find and Fix a Problem	22-3
Step Through File	22-5
End Debugging Session	22-6
Set Breakpoints	22-7
Standard Breakpoints	22-7
Conditional Breakpoints	22-8
Error Breakpoints	22-9
Breakpoints in Anonymous Functions	22-10
Invalid Breakpoints	22-10
Disable Breakpoints	22-10
Clear Breakpoints	22-11
Examine Values While Debugging	22-12
Select Workspace	22-12
View Variable Value	22-12

Publish and Share MATLAB Code	23-2
Create and Share Live Scripts in the Live Editor	23-2
Publish MATLAB Code Files (.m)	23-2
Add Help and Create Documentation	23-4
Publishing Markup	23-6
Markup Overview	23-6
Sections and Section Titles	23-8
Text Formatting	23-9
Bulleted and Numbered Lists	23-10
Text and Code Blocks	23-10
External File Content	23-11
External Graphics	23-12
Image Snapshot	23-14
LaTeX Equations	23-14
Hyperlinks	23-16
HTML Markup	23-18
LaTeX Markup	23-19
Output Preferences for Publishing	23-21
How to Edit Publishing Options	23-21
Specify Output File	23-22
Run Code During Publishing	23-22
Manipulate Graphics in Publishing Output	23-24
Save a Publish Setting	23-27
Manage a Publish Configuration	23-28

Coding and Productivity Tips

Save and Back Up Code	24-2
Save Code	24-2
Back Up Code	24-2
Recommendations on Saving Files	24-3
File Encoding	24-3
Check Code for Errors and Warnings	24-5
Automatically Check Code in the Editor and Live Editor — Code Analyzer	24-5
Create a Code Analyzer Message Report	24-8
Adjust Code Analyzer Message Indicators and Messages	24-8
Understand Code Containing Suppressed Messages	24-11
Understand the Limitations of Code Analysis	24-12
Enable MATLAB Compiler Deployment Messages	24-14
Improve Code Readability	24-16
Indenting Code	24-16
Right-Side Text Limit Indicator	24-17

Code Folding — Expand and Collapse Code Constructs	24-18
Code Refactoring — Automatically convert selected code to a function	24-21
Find and Replace Text in Files	24-22
Find and Replace Any Text in Current File	24-22
Find and Replace Functions or Variables in Current File	24-22
Automatically Rename All Functions or Variables in a File	24-23
Find Text in Multiple File Names or Files	24-24
Function Alternative for Finding Text	24-25
Go To Location in File	24-25
Add Reminders to Files	24-29
Working with TODO/FIXME Reports	24-29
MATLAB Code Analyzer Report	24-31
Running the Code Analyzer Report	24-31
Changing Code Based on Code Analyzer Messages	24-32
Other Ways to Access Code Analyzer Messages	24-32
MATLAB Code Compatibility Report	24-34
Generate the Code Compatibility Report	24-34
Programmatic Use	24-36
Unsupported Functionality	24-36

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	25-2
Protect Your Source Code	25-6
Building a Content Obscured Format with P-Code	25-6
Building a Standalone Executable	25-7
Create Hyperlinks that Run Functions	25-8
Run a Single Function	25-8
Run Multiple Functions	25-9
Provide Command Options	25-9
Include Special Characters	25-9
Create and Share Toolboxes	25-11
Create Toolbox	25-11
Share Toolbox	25-15

Function Argument Validation	26-2
Introduction to Argument Validation	26-2
Where to Use Argument Validation	26-2
arguments Block Syntax	26-2
Examples of Argument Validation	26-5
Kinds of Arguments	26-6
Required and Optional Positional Arguments	26-6
Repeating Arguments	26-8
Name-Value Arguments	26-10
Name-Value Arguments from Class Properties	26-13
Argument Validation in Class Methods	26-15
Order of Argument Validation	26-15
Avoiding Class and Size Conversions	26-16
margin in Argument Validation	26-18
Restrictions on Variable and Function Access	26-19
Debugging Arguments Blocks	26-20
Argument Validation Functions	26-21
Numeric Value Attributes	26-21
Comparison with Other Values	26-22
Data Types	26-22
Size	26-22
Membership and Range	26-23
Text	26-23
Define Validation Functions	26-23
Ways to Parse Function Inputs	26-25
Function Argument Validation	26-25
validateattributes	26-25
inputParser	26-25
Transparency in MATLAB Code	26-26
Writing Transparent Code	26-26

Software Development

Exception Handling in a MATLAB Application	27-2
Overview	27-2
Getting an Exception at the Command Line	27-2
Getting an Exception in Your Program Code	27-3
Generating a New Exception	27-3

Throw an Exception	27-4
Suggestions on How to Throw an Exception	27-4
Respond to an Exception	27-6
Overview	27-6
The try/catch Statement	27-6
Suggestions on How to Handle an Exception	27-7
Clean Up When Functions Complete	27-9
Overview	27-9
Examples of Cleaning Up a Program Upon Exit	27-10
Retrieving Information About the Cleanup Routine	27-11
Using onCleanup Versus try/catch	27-12
onCleanup in Scripts	27-12
Issue Warnings and Errors	27-13
Issue Warnings	27-13
Throw Errors	27-13
Add Run-Time Parameters to Your Warnings and Errors	27-14
Add Identifiers to Warnings and Errors	27-14
SUPPRESS Warnings	27-16
Turn Warnings On and Off	27-16
Restore Warnings	27-18
Disable and Restore a Particular Warning	27-18
Disable and Restore Multiple Warnings	27-19
Change How Warnings Display	27-20
Enable Verbose Warnings	27-20
Display a Stack Trace on a Specific Warning	27-20
Use try/catch to Handle Errors	27-21

Program Scheduling

28

Schedule Command Execution Using Timer	28-2
Overview	28-2
Example: Displaying a Message	28-2
Timer Callback Functions	28-4
Associating Commands with Timer Object Events	28-4
Creating Callback Functions	28-5
Specifying the Value of Callback Function Properties	28-6
Handling Timer Queuing Conflicts	28-8
Drop Mode (Default)	28-8
Error Mode	28-9
Queue Mode	28-10

Measure the Performance of Your Code	29-2
Overview of Performance Timing Functions	29-2
Time Functions	29-2
Time Portions of Code	29-2
The cputime Function vs. tic/toc and timeit	29-2
Tips for Measuring Performance	29-3
Profile Your Code to Improve Performance	29-4
What Is Profiling?	29-4
Profile Your Code	29-4
Profile Multiple Statements in Command Window	29-10
Profile an App	29-11
Determine Code Coverage Using the Profiler	29-12
Techniques to Improve Performance	29-14
Environment	29-14
Code Structure	29-14
Programming Practices for Performance	29-14
Tips on Specific MATLAB Functions	29-15
Preallocation	29-16
Preallocating a Nondouble Matrix	29-16
Vectorization	29-18
Using Vectorization	29-18
Array Operations	29-19
Logical Array Operations	29-20
Matrix Operations	29-21
Ordering, Setting, and Counting Operations	29-22
Functions Commonly Used in Vectorization	29-23

Strategies for Efficient Use of Memory	30-2
Use Appropriate Data Storage	30-2
Avoid Temporary Copies of Data	30-3
Reclaim Used Memory	30-4
Resolve “Out of Memory” Errors	30-6
Leverage tall Arrays	30-6
Leverage the Memory of Multiple Machines	30-7
Load Only as Much Data as You Need	30-7
Increase System Swap Space	30-8
Set the Process Limit on Linux Systems	30-8
Disable Java VM on Linux Systems	30-9

How MATLAB Allocates Memory	30-10
Avoid Unnecessary Copies of Data	30-14
Passing Values to Functions	30-14
Why Pass-by-Value Semantics	30-17
Handle Objects	30-17

Custom Help and Documentation

31

Create Help for Classes	31-2
Help Text from the doc Command	31-2
Custom Help Text	31-3
Check Which Programs Have Help	31-8
Create Help Summary Files — Contents.m	31-10
What Is a Contents.m File?	31-10
Create a Contents.m File	31-10
Check an Existing Contents.m File	31-11
Customize Code Suggestions and Completions	31-12
Function Objects	31-13
Signature Objects	31-13
Argument Objects	31-14
Create Function Signature File	31-17
How Function Signature Information is Used	31-18
Multiple Signatures	31-19
Display Custom Documentation	31-21
Overview	31-21
Create HTML Help Files	31-22
Create info.xml File	31-23
Create helptoc.xml File	31-24
Build a Search Database	31-26
Address Validation Errors for info.xml Files	31-27
Display Custom Examples	31-29
How to Display Examples	31-29
Elements of the demos.xml File	31-30

Projects

32

Create Projects	32-2
What Are Projects?	32-2
Create Project	32-2
Open Project	32-2
Set up Project	32-3

Add Files to Project	32-5
Other Ways to Create Projects	32-6
Automate Startup and Shutdown Tasks	32-8
Specify Project Path	32-8
Set Startup Folder	32-8
Specify Startup and Shutdown Files	32-8
Manage Project Files	32-10
Automatic Updates When Renaming, Deleting, or Removing Files	32-11
Find Project Files	32-12
Group and Sort Project Files	32-12
Search for and Filter Project Files	32-12
Search the Content in Project Files	32-12
Create Shortcuts to Frequent Tasks	32-14
Run Shortcuts	32-14
Create Shortcuts	32-14
Organize Shortcuts	32-14
Add Labels to Project Files	32-16
Add Labels	32-16
View and Edit Label Data	32-16
Create Labels	32-17
Create Custom Tasks	32-18
Create a Custom Task Function	32-18
Run a Custom Task	32-18
Save Custom Task Report	32-19
Componentize Large Projects	32-20
Add or Remove Reference to a Project	32-20
View, Edit, or Run Referenced Project Files	32-20
Extract Folder to Create a Referenced Project	32-21
Manage Changes in Referenced Project Using Checkpoints	32-21
Share Projects	32-23
Create an Export Profile	32-26
Upgrade Projects	32-27
Run Upgrade Project Tool	32-27
Examine Upgrade Project Report	32-28
Analyze Project Dependencies	32-30
Run a Dependency Analysis	32-30
Explore the Dependency Graph, Views, and Filters	32-32
Investigate and Resolve Problems	32-37
Find Required Products and Add-Ons	32-40
Find File Dependencies	32-41
Export Dependency Analysis Results	32-42
Clone from Git Repository	32-44

Use Source Control with Projects	32-45
Setup Source Control	32-45
Perform Source Control Operations	32-47
Work with Derived Files in Projects	32-54
Find Project Files With Unsaved Changes	32-55
Manage Open Files When Closing a Project	32-55
Create and Edit Projects Programmatically	32-56
Explore an Example Project	32-63

Source Control Interface

33

About MathWorks Source Control Integration	33-2
Classic and Distributed Source Control	33-2
Select or Disable Source Control System	33-4
Select Source Control System	33-4
Disable Source Control	33-4
Create New Repository	33-5
Create Git Repository on Your Local System	33-5
Review Changes in Source Control	33-7
Mark Files for Addition to Source Control	33-8
Resolve Source Control Conflicts	33-9
Examining and Resolving Conflicts	33-9
Resolve Conflicts	33-9
Merge Text Files	33-10
Extract Conflict Markers	33-10
Commit Modified Files to Source Control	33-12
Revert Changes in Source Control	33-13
Revert Local Changes	33-13
Revert a File to a Specified Revision	33-13
Set Up SVN Source Control	33-14
SVN Source Control Options	33-14
Register Binary Files with SVN	33-14
Standard Repository Structure	33-17
Tag Versions of Files	33-17
Enforce Locking Files Before Editing	33-17
Share a Subversion Repository	33-18
Check Out from SVN Repository	33-19
Retrieve Tagged Version of Repository	33-19

Update SVN File Status and Revision	33-21
Refresh Status of Files	33-21
Update Revisions of Files	33-21
Get SVN File Locks	33-22
Manage SVN Repository Locks	33-22
Set Up Git Source Control	33-23
Configure MATLAB on Windows	33-23
Use SSH Authentication with MATLAB	33-23
Register Binary Files with Git	33-24
Configure Git Credential Helper	33-25
Add Git Submodules	33-26
Update Submodules	33-26
Use Fetch and Merge with Submodules	33-26
Use Push to Send Changes to the Submodule Repository	33-26
Retrieve Files from Git Repository	33-28
Update Git File Status and Revision	33-29
Refresh Status of Files	33-29
Update Revisions of Files	33-29
Branch and Merge with Git	33-30
Create Branch	33-30
Switch Branch	33-31
Compare Branches	33-32
Merge Branches	33-32
Revert to Head	33-33
Delete Branches	33-33
Pull, Push and Fetch Files with Git	33-34
Pull and Push	33-34
Fetch and Merge	33-35
Use Git Stashes	33-35
Move, Rename, or Delete Files Under Source Control	33-37
Customize External Source Control to Use MATLAB for Diff and Merge	33-38
Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge	33-38
Integration with Git	33-39
Integration with SVN	33-40
Integration with Other Source Control Tools	33-41
MSSCCI Source Control Interface	33-43
Set Up MSSCCI Source Control	33-44
Create Projects in Source Control System	33-44
Specify Source Control System with MATLAB Software	33-45
Register Source Control Project with MATLAB Software	33-46
Add Files to Source Control	33-48

Check Files In and Out from MSSCCI Source Control	33-49
Check Files Into Source Control	33-49
Check Files Out of Source Control	33-49
Undoing the Checkout	33-50
Additional MSSCCI Source Control Actions	33-51
Getting the Latest Version of Files for Viewing or Compiling	33-51
Removing Files from the Source Control System	33-52
Showing File History	33-52
Comparing the Working Copy of a File to the Latest Version in Source Control	33-53
Viewing Source Control Properties of a File	33-54
Starting the Source Control System	33-55
Access MSSCCI Source Control from Editors	33-57
Troubleshoot MSSCCI Source Control Problems	33-58
Source Control Error: Provider Not Present or Not Installed Properly	33-58
Restriction Against @ Character	33-59
Add to Source Control Is the Only Action Available	33-59
More Solutions for Source Control Problems	33-59

34

Unit Testing

Write Test Using Live Script	34-3
Write Script-Based Unit Tests	34-6
Write Script-Based Test Using Local Functions	34-11
Extend Script-Based Tests	34-14
Test Suite Creation	34-14
Test Selection	34-14
Programmatic Access of Test Diagnostics	34-15
Test Runner Customization	34-15
Run Tests in Editor	34-17
Write Function-Based Unit Tests	34-20
Create Test Function	34-20
Run the Tests	34-22
Analyze the Results	34-23
Write Simple Test Case Using Functions	34-24
Write Test Using Setup and Teardown Functions	34-27
Extend Function-Based Tests	34-32
Fixtures for Setup and Teardown Code	34-32
Test Logging and Verbosity	34-33

Test Suite Creation	34-33
Test Selection	34-33
Test Running	34-34
Programmatic Access of Test Diagnostics	34-34
Test Runner Customization	34-35
Author Class-Based Unit Tests in MATLAB	34-36
The Test Class Definition	34-36
The Unit Tests	34-36
Additional Features for Advanced Test Classes	34-37
Write Simple Test Case Using Classes	34-39
Write Setup and Teardown Code Using Classes	34-42
Test Fixtures	34-42
Test Case with Method-Level Setup Code	34-42
Test Case with Class-Level Setup Code	34-43
Table of Verifications, Assertions, and Other Qualifications	34-45
Tag Unit Tests	34-47
Tag Tests	34-47
Select and Run Tests	34-48
Write Tests Using Shared Fixtures	34-51
Create Basic Custom Fixture	34-54
Create Advanced Custom Fixture	34-56
Use Parameters in Class-Based Tests	34-61
How to Write Parameterized Tests	34-61
How to Initialize Parameterization Properties	34-62
Specify Parameterization Level	34-63
Specify How Parameters Are Combined	34-64
Use External Parameters in Tests	34-64
Create Basic Parameterized Test	34-66
Create Advanced Parameterized Test	34-71
Use External Parameters in Parameterized Test	34-78
Define Parameters at Suite Creation Time	34-82
Create Simple Test Suites	34-89
Run Tests for Various Workflows	34-91
Set Up Example Tests	34-91
Run All Tests in Class or Function	34-91
Run Single Test in Class or Function	34-91
Run Test Suites by Name	34-92
Run Test Suites from Test Array	34-92
Run Tests with Customized Test Runner	34-93

Programmatically Access Test Diagnostics	34-94
Add Plugin to Test Runner	34-95
Write Plugins to Extend TestRunner	34-97
Custom Plugins Overview	34-97
Extending Test Session Level Plugin Methods	34-97
Extending Test Suite Level Plugin Methods	34-98
Extending Test Class Level Plugin Methods	34-98
Extending Test Level Plugin Methods	34-99
Create Custom Plugin	34-100
Run Tests in Parallel with Custom Plugin	34-105
Write Plugin to Add Data to Test Results	34-113
Write Plugin to Save Diagnostic Details	34-118
Plugin to Generate Custom Test Output Format	34-122
Analyze Test Case Results	34-125
Analyze Failed Test Results	34-128
Rerun Failed Tests	34-130
Dynamically Filtered Tests	34-133
Test Methods	34-133
Method Setup and Teardown Code	34-135
Class Setup and Teardown Code	34-136
Create Custom Constraint	34-139
Create Custom Boolean Constraint	34-142
Create Custom Tolerance	34-146
Overview of App Testing Framework	34-150
App Testing	34-150
Gesture Support of UI Components	34-150
Write a Test for an App	34-152
Write Test for App	34-155
Write Test That Uses App Testing and Mocking Frameworks	34-159
Create App	34-159
Test App With Manual Intervention	34-160
Create Fully Automated Test	34-161
Overview of Performance Testing Framework	34-164
Determine Bounds of Measured Code	34-164
Types of Time Experiments	34-165
Write Performance Tests with Measurement Boundaries	34-165

Run Performance Tests	34-166
Understand Invalid Test Results	34-166
Test Performance Using Scripts or Functions	34-168
Test Performance Using Classes	34-172
Measure Fast Executing Test Code	34-178
Create Mock Object	34-181
Specify Mock Object Behavior	34-188
Define Mock Method Behavior	34-188
Define Mock Property Behavior	34-189
Define Repeating and Subsequent Behavior	34-190
Summary of Behaviors	34-192
Qualify Mock Object Interaction	34-193
Qualify Mock Method Interaction	34-193
Qualify Mock Property Interaction	34-194
Use Mock Object Constraints	34-195
Summary of Qualifications	34-197
Ways to Write Unit Tests	34-199
Script-Based Unit Tests	34-199
Function-Based Unit Tests	34-200
Class-Based Unit Tests	34-200
Extend Unit Testing Framework	34-201
Compile MATLAB Unit Tests	34-202
Run Tests with Standalone Applications	34-202
Run Tests in Parallel with Standalone Applications	34-203
TestRand Class Definition Summary	34-203
Develop and Integrate Software with Continuous Integration	34-205
Continuous Integration Workflow	34-205
Continuous Integration with MathWorks Products	34-207
Generate Artifacts Using MATLAB Unit Test Plugins	34-209
Continuous Integration with MATLAB on CI Platforms	34-213
Azure DevOps	34-213
CircleCI	34-213
GitHub Actions	34-213
Jenkins	34-213
Travis CI	34-214
Other Platforms	34-214

What Are System Objects?	35-2
Running a System Object	35-3
System Object Functions	35-3
System Objects vs MATLAB Functions	35-5
System Objects vs. MATLAB Functions	35-5
Process Audio Data Using Only MATLAB Functions Code	35-5
Process Audio Data Using System Objects	35-6
System Design in MATLAB Using System Objects	35-7
System Design and Simulation in MATLAB	35-7
Create Individual Components	35-7
Configure Components	35-8
Create and Configure Components at the Same Time	35-8
Assemble Components Into System	35-9
Run Your System	35-9
Reconfiguring Objects	35-10
Define Basic System Objects	35-11
Create System Object Class	35-11
Define Algorithm	35-11
Change the Number of Inputs	35-13
Validate Property and Input Values	35-16
Validate a Single Property	35-16
Validate Interdependent Properties	35-16
Validate Inputs	35-16
Complete Class Example	35-16
Initialize Properties and Setup One-Time Calculations	35-18
Set Property Values at Construction Time	35-20
Reset Algorithm and Release Resources	35-22
Reset Algorithm State	35-22
Release System Object Resources	35-22
Define Property Attributes	35-24
Specify Property as Nontunable	35-24
Specify Property as DiscreteState	35-24
Example Class with Various Property Attributes	35-24
Hide Inactive Properties	35-26
Specify Inactive Property	35-26
Complete Class Definition File with Inactive Properties Method	35-26
Limit Property Values to Finite List	35-28
Property Validation with mustBeMember	35-28
Enumeration Property	35-28
Create a Whiteboard System object	35-29

Process Tuned Properties	35-32
Define Composite System Objects	35-34
Define Finite Source Objects	35-36
Use the FiniteSource Class and Specify End of the Source	35-36
Complete Class Definition File with Finite Source	35-36
Save and Load System Object	35-38
Save System Object and Child Object	35-38
Load System Object and Child Object	35-38
Complete Class Definition Files with Save and Load	35-38
Define System Object Information	35-41
Handle Input Specification Changes	35-43
React to Input Specification Changes	35-43
Restrict Input Specification Changes	35-43
Summary of Call Sequence	35-45
Setup Call Sequence	35-45
Running the Object or Step Call Sequence	35-45
Reset Method Call Sequence	35-46
Release Method Call Sequence	35-47
Detailed Call Sequence	35-48
setup Call Sequence	35-48
Running the Object or step Call Sequence	35-48
reset Call Sequence	35-49
release Call Sequence	35-49
Tips for Defining System Objects	35-50
General	35-50
Inputs and Outputs	35-50
Using ~ as an Input Argument in Method Definitions	35-50
Properties	35-50
Text Comparisons	35-51
Simulink	35-51
Code Generation	35-52
Insert System Object Code Using MATLAB Editor	35-53
Define System Objects with Code Insertion	35-53
Create a Temperature Enumeration	35-55
Create Custom Property for Freezing Point	35-56
Add Method to Validate Inputs	35-57
Analyze System Object Code	35-58
View and Navigate System object Code	35-58
Example: Go to StepImpl Method Using Analyzer	35-58
Use Global Variables in System Objects	35-60
System Object Global Variables in MATLAB	35-60
System Object Global Variables in Simulink	35-60
Create Moving Average System object	35-64

Create New System Objects for File Input and Output	35-69
Create Composite System object	35-75

Language

Syntax Basics

- “Continue Long Statements on Multiple Lines” on page 1-2
- “Ignore Function Outputs” on page 1-3
- “Variable Names” on page 1-4
- “Case and Space Sensitivity” on page 1-5
- “Choose Command Syntax or Function Syntax” on page 1-6
- “Resolve Error: Undefined Function or Variable” on page 1-9

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];
```

Ignore Function Outputs

This example shows how to ignore specific outputs from a function using the tilde (~) operator.

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.

If you do not use the tilde operator, you can request only the first N outputs of a function (where N is less than or equal to the number of possible outputs) and ignore any remaining outputs. For example, request only the first output, ignoring the second and third.

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

If you request more than one output, enclose the variable names in square brackets, []. The following code ignores the output argument `ext`.

```
[helpPath,name] = fileparts(helpFile);
```

To ignore function outputs in any position in the argument list, use the tilde operator. For example, ignore the first output using a tilde.

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

You can ignore any number of function outputs using the tilde operator. Separate consecutive tildes with a comma. For example, this code ignores the first two output arguments.

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

See Also

More About

- “Ignore Inputs in Function Definitions” on page 21-10

Variable Names

In this section...

["Valid Names" on page 1-4](#)

["Conflicts with Function Names" on page 1-4](#)

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`

Examples of 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 2-86

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]
Error using sin
Not enough input arguments.

[1 sin(pi) 3]
ans =
    1.0000    0.0000    3.0000
```

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

Choose Command Syntax or Function Syntax

MATLAB has two ways of calling functions, called function syntax and command syntax. This page discusses the differences between these syntax formats and how to avoid common mistakes associated with command syntax.

For introductory information on calling functions, see “Calling Functions”. For information related to defining functions, see “Create Functions in Files” on page 20-2.

Command Syntax and Function Syntax

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)
```

In function syntax, inputs can be data, variables, and even MATLAB expressions. If an input is data, such as the numeric value 2 or the string array ["a" "b" "c"], MATLAB passes it to the function as-is. If an input is a variable MATLAB will pass the value assigned to it. If an input is an expression, like $2+2$ or $\sin(2*\pi)$, MATLAB evaluates it first, and passes the result to the function. If the function has outputs, you can assign them to variables as shown in the example syntax above.

Command syntax is simpler but more limited. To use it, separate inputs with spaces rather than commas, and do not enclose them in parentheses.

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

With command syntax, MATLAB passes all inputs as character vectors (that is, as if they were enclosed in single quotation marks) and does not assign outputs to variables. To pass a data type other than a character vector, use the function syntax. To pass a value that contains a space, you have two options. One is to use function syntax. The other is to put single quotes around the value. Otherwise, MATLAB treats the space as splitting your value into multiple inputs.

If a value is assigned to 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
open filename	open('filename')	open(filename)
isequal A B	isequal('A','B')	isequal(A,B)
strcmp class(A) int8	strcmp('class(A)', 'int8')	strcmp(class(A), 'int8')
cd tempdir	cd('tempdir')	cd(tempdir)
isnumeric 500	isnumeric('500')	isnumeric(500)
round 3.499	round('3.499'), which is equivalent to round([51 46 52 57 57])	round(3.499)
disp hello world	disp('hello','world')	disp('hello world') or disp 'hello world'
disp "string"	disp('"string"')	disp("string")

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 `'./d'` as its argument. It also could represent element-wise division on the array `ls`, using the variable `d` as the divisor.

If you issue this 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. When you are using those components, 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, MATLAB treats the potentially ambiguous statement `ls ./d` as 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.

See Also

[“Calling Functions”](#) | [“Create Functions in Files”](#) on page 20-2

Resolve Error: Undefined Function or Variable

Issue

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.

Possible Solutions

Verify Spelling of Function or Variable Name

One of the most common causes is misspelling the function or variable name. Especially with longer names or names containing similar characters (such as the letter l and numeral one), it is easy to make mistakes and hard to detect them.

Often, when 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
```

When this happens, press **Enter** to execute the suggested command or **Esc** to dismiss it.

Verify Inputs Correspond to the Function Syntax

Object methods are typically called using function syntax: for instance `method(object, inputs)`. Alternatively, they can be called using dot notation: for instance `object.method(inputs)`. One common error is to mix these syntaxes. For instance, you might call the method using function syntax, but to provide inputs following dot notation syntax and leave out the object as an input: for instance, `method(inputs)`. To avoid this, when calling an object method, make sure you specify the object first, either through the first input of function syntax or through the first identifier of dot notation.

Make Sure Function Name Matches File Name

When you write a function, you establish its name 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`

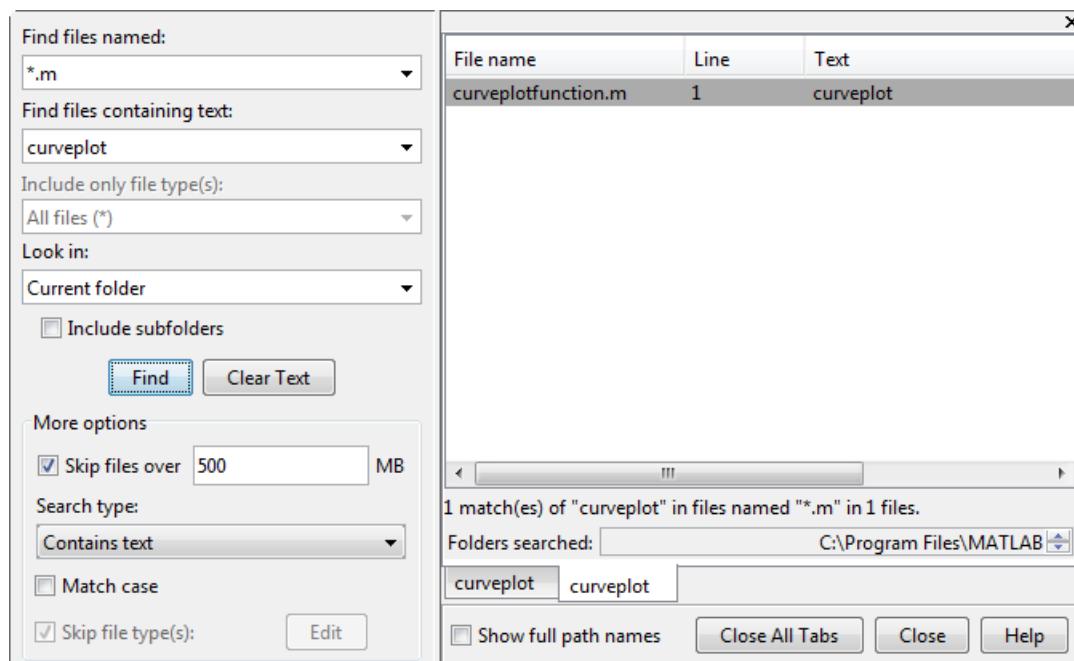
function to a file named `curveplotfunction.m`, then attempts to invoke the function using the function name will fail:

```
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.

To Locate the file that defines 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 Necessary Toolbox Is Installed and Correct Version

If you are unable to use a built-in function from MATLAB or its toolboxes, make sure that the function is installed and is the correct version.

If you do not know which toolbox contains 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. Alternatively, for steps to identify toolboxes that a function depends on, see “Identify Program Dependencies” on page 25-2.

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. If you cannot, you need to install it in order to use it. If

you cannot, you need to install it in order to use it. For help with installing MathWorks products, see “[Installation and Licensing](#)”.

Verify Path Used to Access Function Toolbox

Tip If you have a custom file path, this step will delete it.

The MATLAB search path is a subset of all the folders in the file system. MATLAB uses the search path to locate files used with MathWorks products efficiently. For more information, see “[What Is the MATLAB Search Path?](#)”.

If the function you are attempting to use is part of a toolbox, then verify that the toolbox is available using `ver`.

Because MATLAB stores the toolbox information in a cache file, you need to first update this cache and then reset the path.

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.

The Preference dialog box appears.

- 2 On the **MATLAB > General** page, select **Update Toolbox Path Cache**.

- 3 On the **Home** tab, in the **Environment** section, select  **Set Path**.

The Set Path dialog box opens.

- 4 Select **Default**.

A small dialog box opens warning that you will lose your current path settings if you proceed. Select **Yes** if you decide to proceed.

Run `ver` to see if the toolbox is installed. If not, you may need to reinstall this toolbox to use this function. For more information about installing a toolbox, see [How do I install additional toolboxes into an existing installation of MATLAB](#).

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. If MATLAB finds your function file, it presents you with the path to it. You can add that file to the path using the `addpath` function. If it does not, make sure the necessary toolbox is installed, and that it is the correct version.

Confirm The License Is Active

If you are unable to use a built-in function from a MATLAB toolbox and have confirmed that the toolbox is installed, make sure that you have an active license for that toolbox. Use `license` to display currently active licenses. For additional support for managing licenses, see “[Manage Your Licenses](#)”.

Program Components

- “MATLAB Operators and Special Characters” on page 2-2
- “Array vs. Matrix Operations” on page 2-20
- “Compatible Array Sizes for Basic Operations” on page 2-25
- “Array Comparison with Relational Operators” on page 2-29
- “Operator Precedence” on page 2-32
- “Average Similar Data Points Using a Tolerance” on page 2-34
- “Group Scattered Data Using a Tolerance” on page 2-36
- “Bit-Wise Operations” on page 2-38
- “Perform Cyclic Redundancy Check” on page 2-44
- “Conditional Statements” on page 2-47
- “Loop Control Statements” on page 2-49
- “Regular Expressions” on page 2-51
- “Lookahead Assertions in Regular Expressions” on page 2-63
- “Tokens in Regular Expressions” on page 2-66
- “Dynamic Regular Expressions” on page 2-72
- “Comma-Separated Lists” on page 2-79
- “Alternatives to the eval Function” on page 2-86

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	mrddivide
.\	Element-wise left division	ldivide
\	Matrix left division (also known as <i>backslash</i>)	mlddivide
.^	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
>=	Greater than or equal to	ge
<	Less than	lt
<=	Less than or equal to	le

Logical Operators

Symbol	Role	More Information
&	Find logical AND	and
	Find logical OR	or
&&	Find logical AND (with short-circuiting)	Logical Operators: Short-Circuit &&

Symbol	Role	More Information
	Find logical OR (with short-circuiting)	
~	Find logical NOT	not

Special Characters

@	<p>Name: At symbol</p> <p>Uses:</p> <ul style="list-style-type: none"> • Function handle construction and reference • Calling superclass methods <p>Description: The @ symbol forms a handle to either the named function that follows the @ sign, or to the anonymous function that follows the @ sign. You can also use @ to call superclass methods from subclasses.</p> <p>Examples</p> <p>Create a function handle to a named function:</p> <pre>fhandle = @myfun</pre> <p>Create a function handle to an anonymous function:</p> <pre>fhandle = @(x,y) x.^2 + y.^2;</pre> <p>Call the <code>disp</code> method of <code>MySuper</code> from a subclass:</p> <pre>disp@MySuper(obj)</pre> <p>Call the superclass constructor from a subclass using the object being constructed:</p> <pre>obj = obj@MySuper(arg1,arg2,...)</pre> <p>More Information:</p> <ul style="list-style-type: none"> • “Create Function Handle” on page 13-2 • “Call Superclass Methods on Subclass Objects”
---	--

.	<p>Name: Period or dot</p> <p>Uses:</p> <ul style="list-style-type: none">• Decimal point• Element-wise operations• Structure field access• Object property or method specifier <p>Description: The period character separates the integral and fractional parts of a number, such as 3.1415. MATLAB operators that contain a period always work element-wise. The period character also enables you to access the fields in a structure, as well as the properties and methods of an object.</p> <p>Examples</p> <p>Decimal point: 102.5543</p> <p>Element-wise operations: A.*B A.^2</p> <p>Structure field access: myStruct.f1</p> <p>Object property specifier: myObj.PropertyName</p> <p>More Information</p> <ul style="list-style-type: none">• “Array vs. Matrix Operations” on page 2-20• “Structures”• “Access Property Values”
---	---

...	<p>Name: Dot dot dot or ellipsis</p> <p>Uses: Line continuation</p> <p>Description: Three or more periods at the end of a line continues the current command on the next line. If three or more periods occur before the end of a line, then MATLAB ignores the rest of the line and continues to the next line. This effectively makes a comment out of anything on the current line that follows the three periods.</p> <hr/> <p>Note MATLAB interprets the ellipsis as a space character. Therefore, multi-line commands must be valid as a single line with the ellipsis replaced by a space character.</p> <p>Examples</p> <p>Continue a function call on the next line:</p> <pre>sprintf(['The current value ' ... 'of %s is %d'],vname,value)</pre> <p>Break a character vector up on multiple lines and concatenate the lines together:</p> <pre>S = ['If three or more periods occur before the '... 'end of a line, then the rest of that line is ' ... 'ignored and MATLAB continues to the next line']</pre> <p>To comment out one line in a multiline command, use <code>...</code> at the beginning of the line to ensure that the command remains complete. If you use <code>%</code> to comment out a line it produces an error:</p> <pre>y = 1 +... 2 +... % 3 +... 4;</pre> <p>However, this code runs properly since the third line does not produce a gap in the command:</p> <pre>y = 1 +... 2 +... ... 3 +... 4;</pre> <p>More Information</p> <ul style="list-style-type: none"> “Continue Long Statements on Multiple Lines” on page 1-2
-----	---

,	<p>Name: Comma</p> <p>Uses: Separator</p> <p>Description: Use commas to separate row elements in an array, array subscripts, function input and output arguments, and commands entered on the same line.</p> <p>Examples</p> <p>Separate row elements to create an array:</p> <pre>A = [12,13; 14,15]</pre> <p>Separate subscripts:</p> <pre>A(1,2)</pre> <p>Separate input and output arguments in function calls:</p> <pre>[Y,I] = max(A,[],2)</pre> <p>Separate multiple commands on the same line (showing output):</p> <pre>figure, plot(sin(-pi:0.1:pi)), grid on</pre> <p>More Information</p> <ul style="list-style-type: none">• horzcat
---	--

:	<p>Name: Colon</p> <p>Uses:</p> <ul style="list-style-type: none">• Vector creation• Indexing• For-loop iteration <p>Description: Use the colon operator to create regularly spaced vectors, index into arrays, and define the bounds of a <code>for</code> loop.</p> <p>Examples</p> <p>Create a vector:</p> <pre>x = 1:10</pre> <p>Create a vector that increments by 3:</p> <pre>x = 1:3:19</pre> <p>Reshape a matrix into a column vector:</p> <pre>A(:)</pre> <p>Assign new elements without changing the shape of an array:</p> <pre>A = rand(3,4); A(:) = 1:12;</pre> <p>Index a range of elements in a particular dimension:</p> <pre>A(2:5,3)</pre> <p>Index all elements in a particular dimension:</p> <pre>A(:,3)</pre> <p>for loop bounds:</p> <pre>x = 1; for k = 1:25 x = x + x^2; end</pre> <p>More Information</p> <ul style="list-style-type: none">• colon• “Creating, Concatenating, and Expanding Matrices”
---	---

;	<p>Name: Semicolon</p> <p>Uses:</p> <ul style="list-style-type: none">• Signify end of row• Suppress output of code line <p>Description: Use semicolons to separate rows in an array creation command, or to suppress the output display of a line of code.</p> <p>Examples</p> <p>Separate rows to create an array:</p> <pre>A = [12,13; 14,15]</pre> <p>Suppress code output:</p> <pre>Y = max(A);</pre> <p>Separate multiple commands on a single line (suppressing output):</p> <pre>A = 12.5; B = 42.7, C = 1.25; B = 42.7000</pre> <p>More Information</p> <ul style="list-style-type: none">• vertcat
---	--

()	<p>Name: Parentheses</p> <p>Uses:</p> <ul style="list-style-type: none">• Operator precedence• Function argument enclosure• Indexing <p>Description: Use parentheses to specify precedence of operations, enclose function input arguments, and index into an array.</p> <p>Examples</p> <p>Precedence of operations:</p> <pre>(A.*(B./C)) - D</pre> <p>Function argument enclosure:</p> <pre>plot(X,Y, 'r*') C = union(A,B)</pre> <p>Indexing:</p> <pre>A(3,:) A(1,2) A(1:5,1)</pre> <p>More Information</p> <ul style="list-style-type: none">• “Operator Precedence” on page 2-32• “Array Indexing”
-----	--

[]	<p>Name: Square brackets</p> <p>Uses:</p> <ul style="list-style-type: none">• Array construction• Array concatenation• Empty matrix and array element deletion• Multiple output argument assignment <p>Description: Square brackets enable array construction and concatenation, creation of empty matrices, deletion of array elements, and capturing values returned by a function.</p> <p>Examples</p> <p>Construct a three-element vector:</p> <pre>X = [10 12 -3]</pre> <p>Add a new bottom row to a matrix:</p> <pre>A = rand(3); A = [A; 10 20 30]</pre> <p>Create an empty matrix:</p> <pre>A = []</pre> <p>Delete a matrix column:</p> <pre>A(:,1) = []</pre> <p>Capture three output arguments from a function:</p> <pre>[C,iA,iB] = union(A,B)</pre> <p>More Information</p> <ul style="list-style-type: none">• “Creating, Concatenating, and Expanding Matrices”• <code>horzcat</code>• <code>vertcat</code>
-----	---

{ }	<p>Name: Curly brackets</p> <p>Uses: Cell array assignment and contents</p> <p>Description: Use curly braces to construct a cell array, or to access the contents of a particular cell in a cell array.</p> <p>Examples</p> <p>To construct a cell array, enclose all elements of the array in curly braces:</p> <pre>C = {[2.6 4.7 3.9], rand(8)*6, 'C. Coolidge'}</pre> <p>Index to a specific cell array element by enclosing all indices in curly braces:</p> <pre>A = C{4,7,2}</pre> <p>More Information</p> <ul style="list-style-type: none"> “Cell Arrays”
%	<p>Name: Percent</p> <p>Uses:</p> <ul style="list-style-type: none"> Comment Conversion specifier <p>Description: The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code.</p> <p>Some functions also interpret the percent sign as a conversion specifier.</p> <p>Two percent signs, %%, serve as a cell delimiter as described in “Code Sections” on page 18-5.</p> <p>Examples</p> <p>Add a comment to a block of code:</p> <pre>% The purpose of this loop is to compute % the value of ...</pre> <p>Use conversion specifier with sprintf:</p> <pre>sprintf('%s = %d', name, value)</pre> <p>More Information</p> <ul style="list-style-type: none"> “Add Comments to Programs” on page 18-3

<p><code>%{ %}</code></p>	<p>Name: Percent curly bracket</p> <p>Uses: Block comments</p> <p>Description: The <code>%{</code> and <code>%}</code> symbols enclose a block of comments that extend beyond one line.</p> <hr/> <p>Note With the exception of whitespace characters, the <code>%{</code> and <code>%}</code> 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.</p> <p>Examples</p> <p>Enclose any multiline comments with percent followed by an opening or closing brace:</p> <pre>%{ The purpose of this routine is to compute the value of ... %}</pre> <p>More Information</p> <ul style="list-style-type: none"> “Add Comments to Programs” on page 18-3
<p>!</p>	<p>Name: Exclamation point</p> <p>Uses: Operating system command</p> <p>Description: The exclamation point precedes operating system commands that you want to execute from within MATLAB.</p> <p>Not available in MATLAB Online™.</p> <p>Examples</p> <p>The exclamation point initiates a shell escape function. Such a function is to be performed directly by the operating system:</p> <pre>!rmdir oldtests</pre> <p>More Information</p> <ul style="list-style-type: none"> “Shell Escape Function Example”

?	<p>Name: Question mark</p> <p>Uses: Metaclass for MATLAB class</p> <p>Description: The question mark retrieves the <code>meta.class</code> object for a particular class name. The ? operator works only with a class name, not an object.</p> <p>Examples</p> <p>Retrieve the meta.class object for class <code>inputParser</code>:</p> <pre>?inputParser</pre> <p>More Information</p> <ul style="list-style-type: none"> • metaclass
''	<p>Name: Single quotes</p> <p>Uses: Character array constructor</p> <p>Description: Use single quotes to create character vectors that have class <code>char</code>.</p> <p>Examples</p> <p>Create a character vector:</p> <pre>chr = 'Hello, world'</pre> <p>More Information</p> <ul style="list-style-type: none"> • “Text in String and Character Arrays” on page 6-2
'''	<p>Name: Double quotes</p> <p>Uses: String constructor</p> <p>Description: Use double quotes to create string scalars that have class <code>string</code>.</p> <p>Examples</p> <p>Create a string scalar:</p> <pre>S = "Hello, world"</pre> <p>More Information</p> <ul style="list-style-type: none"> • “Text in String and Character Arrays” on page 6-2

N/A	<p>Name: Space character</p> <p>Uses: Separator</p> <p>Description: Use the space character to separate row elements in an array constructor, or the values returned by a function. In these contexts, the space character and comma are equivalent.</p> <p>Examples</p> <p>Separate row elements to create an array:</p> <pre>% These statements are equivalent A = [12 13; 14 15] A = [12,13; 14,15]</pre> <p>Separate output arguments in function calls:</p> <pre>% These statements are equivalent [Y I] = max(A) [Y,I] = max(A)</pre>
N/A	<p>Name: Newline character</p> <p>Uses: Separator</p> <p>Description: Use the newline character to separate rows in an array construction statement. In that context, the newline character and semicolon are equivalent.</p> <p>Examples</p> <p>Separate rows in an array creation command:</p> <pre>% These statements are equivalent A = [12 13 14 15] A = [12 13; 14 15]</pre>

~	<p>Name: Tilde</p> <p>Uses:</p> <ul style="list-style-type: none"> • Logical NOT • Argument placeholder <p>Description: Use the tilde symbol to represent logical NOT or to suppress specific input or output arguments.</p> <p>Examples</p> <p>Calculate the logical NOT of a matrix:</p> <pre>A = eye(3); ~A</pre> <p>Determine where the elements of A are not equal to those of B:</p> <pre>A = [1 -1; 0 1] B = [1 -2; 3 2] A~=B</pre> <p>Return only the third output value of union:</p> <pre>[~,~,iB] = union(A,B)</pre> <p>More Information</p> <ul style="list-style-type: none"> • not • “Ignore Inputs in Function Definitions” on page 21-10 • “Ignore Function Outputs” on page 1-3
=	<p>Name: Equal sign</p> <p>Uses: Assignment</p> <p>Description: Use the equal sign to assign values to a variable. The syntax <code>B = A</code> stores the elements of A in variable B.</p> <p>Note The = character is for assignment, whereas the == character is for comparing the elements in two arrays. See <code>eq</code> for more information.</p> <p>Examples</p> <p>Create a matrix A. Assign the values in A to a new variable, B. Lastly, assign a new value to the first element in B.</p> <pre>A = [1 0; -1 0]; B = A; B(1) = 200;</pre>

< &	<p>Name: Left angle bracket and ampersand</p> <p>Uses: Specify superclasses</p> <p>Description: Specify one or more superclasses in a class definition</p> <p>Examples</p> <p>Define a class that derives from one superclass:</p> <pre>classdef MyClass < MySuperclass ... end</pre> <p>Define a class that derives from multiple superclasses:</p> <pre>classdef MyClass < Superclass1 & Superclass2 & end</pre> <p>More Information:</p> <ul style="list-style-type: none"> “Subclass Syntax”
. ?	<p>Name: Dot question mark</p> <p>Uses: Specify fields of name-value structure</p> <p>Description:</p> <p>When using function argument validation, you can define the fields of the name-value structure as the names of all writeable properties of the class.</p> <p>Examples</p> <p>Specify the field names of the <code>propArgs</code> structure as the writeable properties of the <code>matlab.graphics.primitive.Line</code> class.</p> <pre>function f(propArgs) arguments propArgs.?matlab.graphics.primitive.Line end % Function code ... end</pre> <p>More Information:</p> <ul style="list-style-type: none"> “Name-Value Arguments from Class Properties” on page 26-13

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.

/ \	<p>Name: Slash and Backslash</p> <p>Uses: File or folder path separation</p> <p>Description: In addition to their use as mathematical operators, the slash and backslash characters separate the elements of a path or folder. On Microsoft Windows based systems, both slash and backslash have the same effect. On The Open Group UNIX based systems, you must use slash only.</p> <p>Examples</p> <p>On a Windows system, you can use either backslash or slash:</p> <pre>dir([matlabroot '\toolbox\matlab\elmat\shiftdim.m']) dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])</pre> <p>On a UNIX system, use only the forward slash:</p> <pre>dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])</pre>
..	<p>Name: Dot dot</p> <p>Uses: Parent folder</p> <p>Description: Two dots in succession refers to the parent of the current folder. Use this character to specify folder paths relative to the current folder.</p> <p>Examples</p> <p>To go up two levels in the folder tree and down into the <code>test</code> folder, use:</p> <pre>cd ...\\..\\test</pre> <p>More Information</p> <ul style="list-style-type: none"> • <code>cd</code>
*	<p>Name: Asterisk</p> <p>Uses: Wildcard character</p> <p>Description: In addition to being the symbol for matrix multiplication, the asterisk * is used as a wildcard character.</p> <p>Wildcards are generally used in file operations that act on multiple files or folders. MATLAB matches all characters in the name exactly except for the wildcard character *, which can match any one or more characters.</p> <p>Examples</p> <p>Locate all files with names that start with <code>january_</code> and have a <code>.mat</code> file extension:</p> <pre>dir('january_*.*.mat')</pre>

@	<p>Name: At symbol</p> <p>Uses: Class folder indicator</p> <p>Description: An @ sign indicates the name of a class folder.</p> <p>Examples</p> <p>Refer to a class folder:</p> <pre>\@myClass\get.m</pre> <p>More Information</p> <ul style="list-style-type: none"> “Class and Path Folders”
+	<p>Name: Plus</p> <p>Uses: Package directory indicator</p> <p>Description: A + sign indicates the name of a package folder.</p> <p>Examples</p> <p>Package folders always begin with the + character:</p> <pre>+mypack +mypack/pkfcn.m % a package function +mypack/@myClass % class folder in a package</pre> <p>More Information</p> <ul style="list-style-type: none"> “Packages Create Namespaces”

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 6-24.

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
\v	Vertical tab
\xN	Hexadecimal number, N

Symbol	Effect on Text
\N	Octal number, N

See Also

More About

- “Array vs. Matrix Operations” on page 2-20
- “Array Comparison with Relational Operators” on page 2-29
- “Compatible Array Sizes for Basic Operations” on page 2-25
- “Operator Precedence” on page 2-32
- “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 2-20](#)

["Array Operations" on page 2-20](#)

["Matrix Operations" on page 2-22](#)

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 2-25.

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 =
```

$$\begin{matrix} 1 & 3 \\ 2 & 4 \end{matrix}$$

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

```
B =
```

$$\begin{matrix} 3 & 0 \\ 1 & 5 \end{matrix}$$

```
A*B
```

```
ans =
```

$$\begin{matrix} 6 & 15 \\ 10 & 20 \end{matrix}$$

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

```
A.*B
```

```
ans =
```

$$\begin{matrix} 3 & 0 \\ 2 & 20 \end{matrix}$$

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\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'\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

Operator	Purpose	Description	Reference Page
'	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 2-25
- “MATLAB Operators and Special Characters” on page 2-2
- “Operator Precedence” on page 2-32

Compatible Array Sizes for Basic Operations

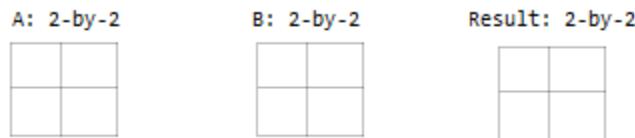
Most binary (two-input) operators and functions in MATLAB support numeric arrays that have *compatible sizes*. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1. In the simplest cases, two array sizes are compatible if 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

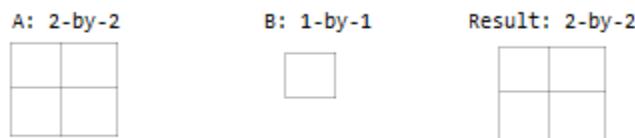
2-D Inputs

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

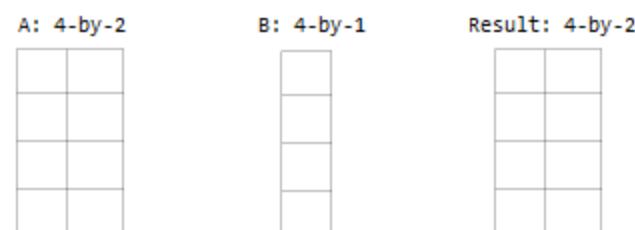
- Two inputs which are exactly the same size.



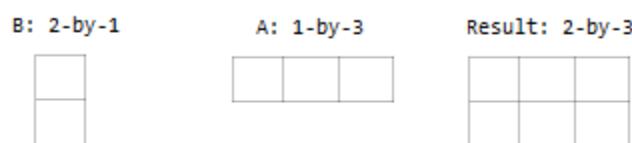
- One input is a scalar.



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



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

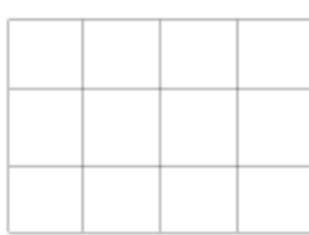


Multidimensional Arrays

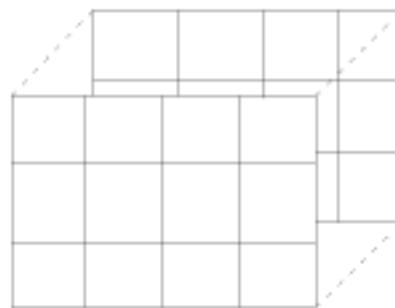
Every array in MATLAB has trailing dimensions of size 1. For multidimensional arrays, this means that a 3-by-4 matrix is the same as a matrix of size 3-by-4-by-1-by-1-by-1. Examples of multidimensional arrays with compatible sizes are:

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

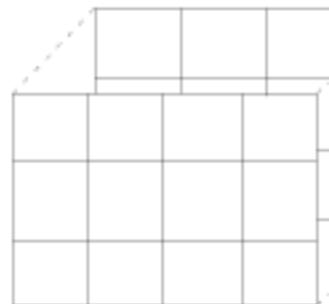
A: 3-by-4



B: 3-by-4-by-2

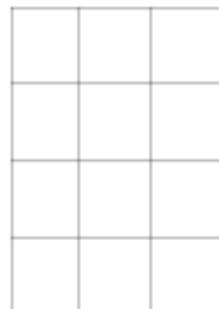


Result: 3-by-4-by-2

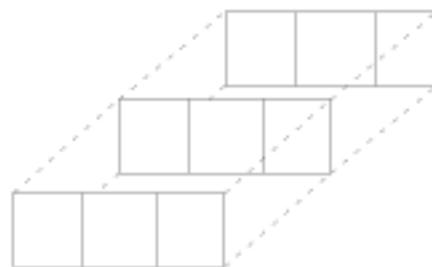


- One input is a matrix, and the other is a 3-D array. The dimensions are all either the same or one of them is 1.

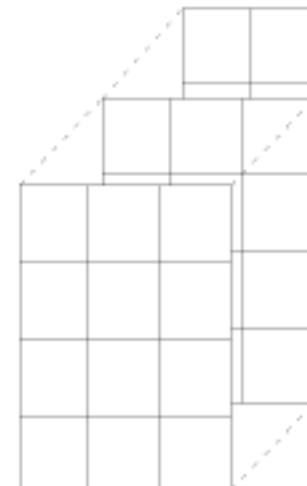
A: 4-by-3



B: 1-by-3-by-3



Result: 4-by-3



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 2-20
- “MATLAB Operators and Special Characters” on page 2-2

Array Comparison with Relational Operators

In this section...

["Array Comparison" on page 2-29](#)

["Logic Statements" on page 2-31](#)

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 2-25.

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), `xor(A,B)` (XOR), and `~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 2-20
- “Compatible Array Sizes for Basic Operations” on page 2-25
- “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

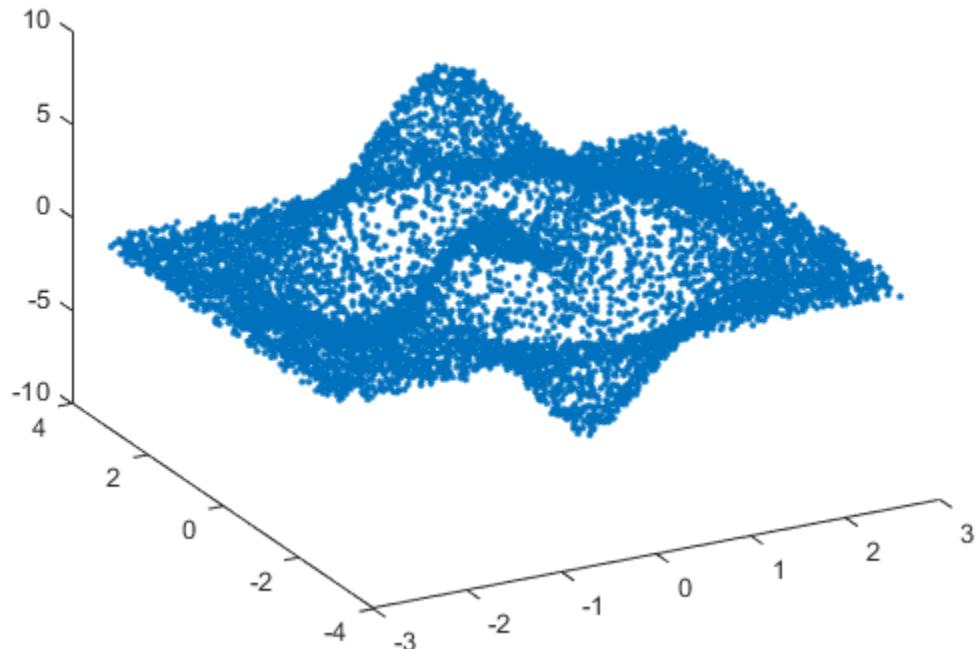
- “Array vs. Matrix Operations” on page 2-20
- “Compatible Array Sizes for Basic Operations” on page 2-25
- “Array Comparison with Relational Operators” on page 2-29
- “MATLAB Operators and Special Characters” on page 2-2

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), '.*')
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.

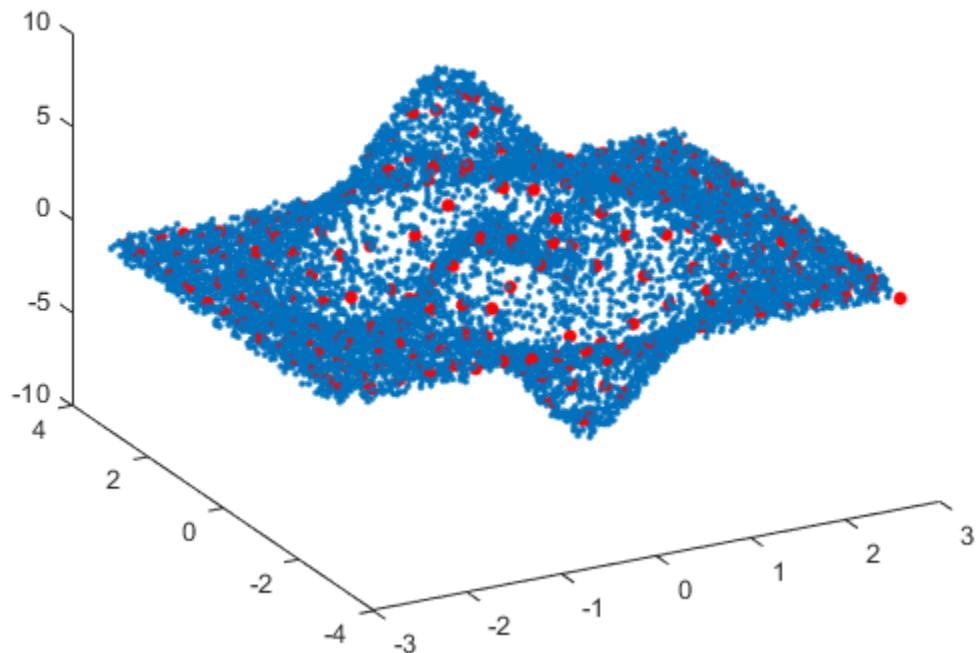
```
DS = [1 1 Inf];
[C,ia] = uniqueTol(A, 0.3, 'ByRows', true, ...
    'OutputAllIndices', true, 'DataScale', DS);
```

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

[unique](#)
[tol](#)

More About

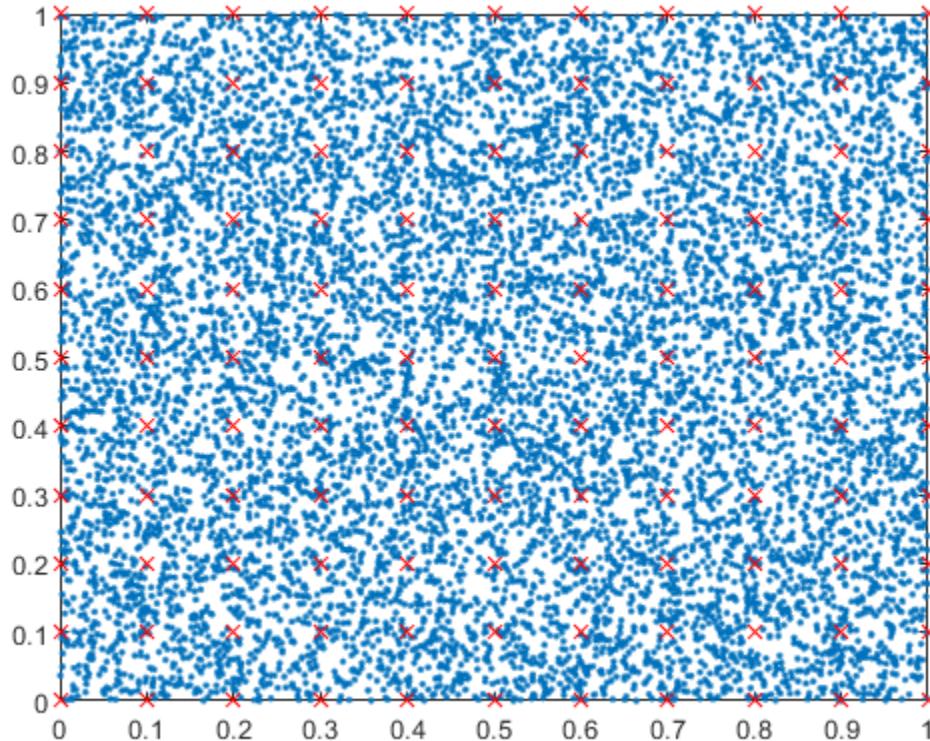
- “Group Scattered Data Using a Tolerance” on page 2-36

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)
```



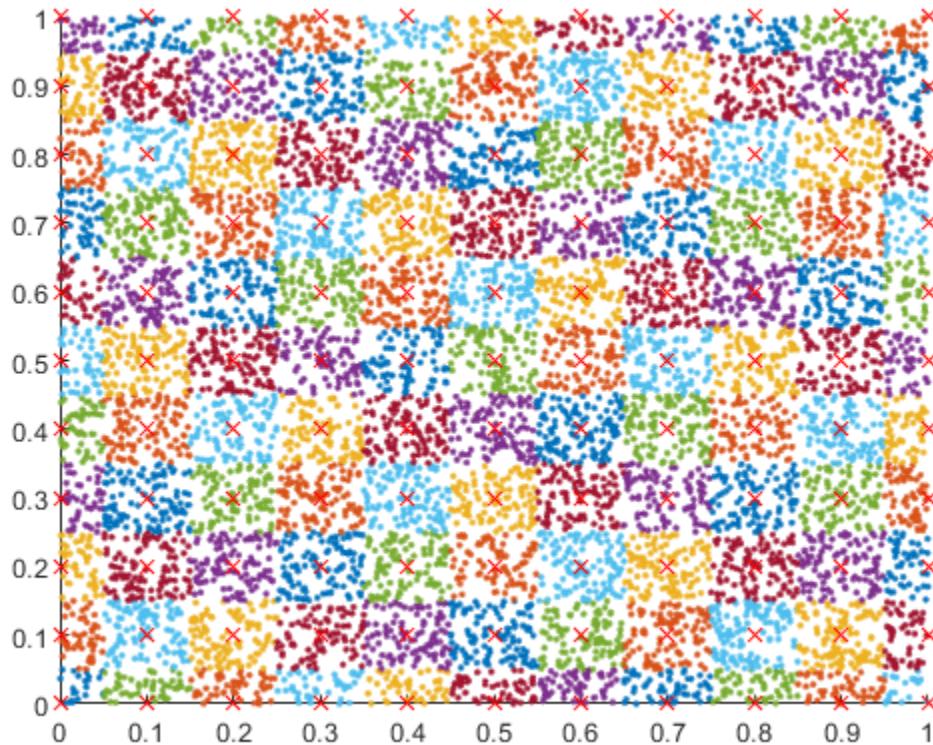
Use `ismembertol` to locate the data points in `x` that are within tolerance of the grid points in `gridPoints`. Use these options with `ismembertol`:

- 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 2-34

Bit-Wise Operations

This topic shows how to use bit-wise operations in MATLAB® to manipulate the bits of numbers. Operating on bits is directly supported by most modern CPUs. In many cases, manipulating the bits of a number in this way is quicker than performing arithmetic operations like division or multiplication.

Number Representations

Any number can be represented with bits (also known as *binary digits*). The binary, or base 2, form of a number contains 1s and 0s to indicate which powers of 2 are present in the number. For example, the 8-bit binary form of 7 is

```
00000111
```

A collection of 8 bits is also called 1 *byte*. In binary representations, the bits are counted from the right to the left, so the first bit in this representation is a 1. This number represents 7 because

$$2^2 + 2^1 + 2^0 = 7.$$

When you type numbers into MATLAB, it assumes the numbers are double precision (a 64-bit binary representation). However, you can also specify single-precision numbers (32-bit binary representation) and integers (signed or unsigned, from 8 to 64 bits). For example, the most memory efficient way to store the number 7 is with an 8-bit unsigned integer:

```
a = uint8(7)  
a = uint8  
    7
```

You can even specify the binary form directly using the prefix `0b` followed by the binary digits (for more information, see “Hexadecimal and Binary Values” on page 6-55). MATLAB stores the number in an integer format with the fewest number of bits. Instead of specifying all the bits, you need to specify only the left-most 1 and all the digits to the right of it. The bits to the left of that bit are trivially zero. So the number 7 is:

```
b = 0b111  
b = uint8  
    7
```

MATLAB stores negative integers using two's complement. For example, consider the 8-bit signed integer -8. To find the two's complement bit pattern for this number:

- 1 Start with the bit pattern of the positive version of the number, 8: `00001000`.
- 2 Next, flip all of the bits: `11110111`.
- 3 Finally, add 1 to the result: `11111000`.

The result, `11111000`, is the bit pattern for -8:

```
n = 0b11111000s8  
n = int8  
    -8
```

MATLAB does not natively display the binary format of numbers. For that, you can use the `dec2bin` function, which returns a character vector of binary digits for positive integers. Again, this function returns only the digits that are not trivially zero.

```
dec2bin(b)
```

```
ans =
'111'
```

You can use `bin2dec` to switch between the two formats. For example, you can convert the binary digits `10110101` to decimal format with the commands

```
data = [1 0 1 1 0 1 0 1];
dec = bin2dec(num2str(data))

dec = 181
```

The `cast` and `typecast` functions are also useful to switch among different data types. These functions are similar, but they differ in how they treat the underlying storage of the number:

- `cast` — Changes the underlying data type of a variable.
- `typecast` — Converts data types without changing the underlying bits.

Because MATLAB does not display the digits of a binary number directly, you must pay attention to data types when you work with bit-wise operations. Some functions return binary digits as a character vector (`dec2bin`), some return the decimal number (`bitand`), and others return a vector of the bits themselves (`bitget`).

Bit Masking with Logical Operators

MATLAB has several functions that enable you to perform logical operations on the bits of two equal-length binary representations of numbers, known as *bit masking*:

- `bitand` — If *both* digits are 1, then the resulting digit is also a 1. Otherwise, the resulting digit is 0.
- `bitor` — If *either* digit is 1, then the resulting digit is also a 1. Otherwise, the resulting digit is 0.
- `bitxor` — If the digits are different, then the resulting digit is a 1. Otherwise, the resulting digit is 0.

In addition to these functions, the bit-wise complement is available with `bitcmp`, but this is a unary operation that flips the bits in only one number at a time.

One use of bit masking is to query the status of a particular bit. For example, if you use a bit-wise AND operation with the binary number `00001000`, you can query the status of the fourth bit. You can then shift that bit to the first position so that MATLAB returns a 0 or 1 (the next section describes bit shifting in more detail).

```
n = 0b10111001;
n4 = bitand(n, 0b1000);
n4 = bitshift(n4, -3)

n4 = uint8
1
```

Bit-wise operations can have surprising applications. For example, consider the 8-bit binary representation of the number $n = 8$:

```
00001000
```

8 is a power of 2, so its binary representation contains a single 1. Now consider the number $(n - 1) = 7$:

```
00000111
```

By subtracting 1, all of the bits starting at the right-most 1 are flipped. As a result, when n is a power of 2, corresponding digits of n and $(n - 1)$ are always different, and the bit-wise AND returns zero.

```
n = 0b1000;  
bitand(n,n-1)  
  
ans = uint8  
     0
```

However, when n is not a power of 2, then the right-most 1 is for the 2^0 bit, so n and $(n - 1)$ have all the same bits except for the 2^0 bit. For this case, the bit-wise AND returns a nonzero number.

```
n = 0b101;  
bitand(n,n-1)  
  
ans = uint8  
     4
```

This operation suggests a simple function that operates on the bits of a given input number to check whether the number is a power of 2:

```
function tf = isPowerOfTwo(n)  
    tf = n && ~bitand(n,n-1);  
end
```

The use of the short-circuit AND operator `&&` checks to make sure that `n` is not zero. If it is, then the function does not need to calculate `bitand(n,n-1)` to know that the correct answer is `false`.

Shifting Bits

Because bit-wise logical operations compare corresponding bits in two numbers, it is useful to be able to move the bits around to change which bits are compared. You can use `bitshift` to perform this operation:

- `bitshift(A,N)` shifts the bits of `A` to the *left* by `N` digits. This is equivalent to *multiplying A by* 2^N .
- `bitshift(A,-N)` shifts the bits of `A` to the *right* by `N` digits. This is equivalent to *dividing A by* 2^N .

These operations are sometimes written `A<<N` (left shift) and `A>>N` (right shift), but MATLAB does not use `<<` and `>>` operators for this purpose.

When the bits of a number are shifted, some bits fall off the end of the number, and 0s or 1s are introduced to fill in the newly created space. When you shift bits to the left, the bits are filled in on the right; when you shift bits to the right, the bits are filled in on the left.

For example, if you shift the bits of the number 8 (binary: 1000) to the right by one digit, you get 4 (binary: 100).

```
n = 0b1000;
bitshift(n,-1)

ans = uint8
    4
```

Similarly, if you shift the number 15 (binary: 1111) to the left by two digits, you get 60 (binary: 111100).

```
n = 0b1111;
bitshift(15,2)

ans = 60
```

When you shift the bits of a negative number, `bitshift` preserves the signed bit. For example, if you shift the signed integer -3 (binary: 11111101) to the right by 2 digits, you get -1 (binary: 11111111). In these cases, `bitshift` fills in on the left with 1s rather than 0s.

```
n = 0b11111101s8;
bitshift(n,-2)

ans = int8
    -1
```

Writing Bits

You can use the `bitset` function to change the bits in a number. For example, change the first bit of the number 8 to a 1 (which adds 1 to the number):

```
bitset(8,1)

ans = 9
```

By default, `bitset` flips bits to *on* or 1. You can optionally use the third input argument to specify the bit value.

`bitset` does not change multiple bits at once, so you need to use a `for` loop to change multiple bits. Therefore, the bits you change can be either consecutive or nonconsecutive. For example, change the first two bits of the binary number 1000:

```
bits = [1 2];
c = 0b1000;
for k = 1:numel(bits)
    c = bitset(c,bits(k));
end
dec2bin(c)

ans =
'1011'
```

Another common use of `bitset` is to convert a vector of binary digits into decimal format. For example, use a loop to set the individual bits of the integer 11001101.

```
data = [1 1 0 0 1 1 0 1];
n = length(data);
dec = 0b0u8;
for k = 1:n
    dec = bitset(dec,n+1-k,data(k));
```

```
end  
dec  
  
dec = uint8  
    205  
  
dec2bin(dec)  
  
ans =  
'11001101'
```

Reading Consecutive Bits

Another use of bit shifting is to isolate consecutive sections of bits. For example, read the last four bits in the 16-bit number `0110000010100000`. Recall that the last four bits are on the *left* of the binary representation.

```
n = 0b0110000010100000;  
dec2bin(bitshift(n,-12))  
  
ans =  
'110'
```

To isolate consecutive bits in the middle of the number, you can combine the use of bit shifting with logical masking. For example, to extract the 13th and 14th bits, you can shift the bits to the right by 12 and then mask the resulting four bits with `0011`. Because the inputs to `bitand` must be the same integer data type, you can specify `0011` as an unsigned 16-bit integer with `0b11u16`. Without the `-u16` suffix, MATLAB stores the number as an unsigned 8-bit integer.

```
m = 0b11u16;  
dec2bin(bitand(bitshift(n,-12),m))  
  
ans =  
'10'
```

Another way to read consecutive bits is with `bitget`, which reads specified bits from a number. You can use colon notation to specify several consecutive bits to read. For example, read the last 8 bits of `n`.

```
bitget(n,16:-1:8)  
  
ans = 1x9 uint16 row vector  
     0   1   1   0   0   0   0   0   1
```

Reading Nonconsecutive Bits

You can also use `bitget` to read bits from a number when the bits are not next to each other. For example, read the 5th, 8th, and 14th bits from `n`.

```
bits = [14 8 5];  
bitget(n,bits)  
  
ans = 1x3 uint16 row vector
```

1 1 0

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitshift` | `bitxor`

More About

- “Integers” on page 4-2
- “Perform Cyclic Redundancy Check” on page 2-44
- “Hexadecimal and Binary Values” on page 6-55

Perform Cyclic Redundancy Check

This example shows how to perform a *cyclic redundancy check* (CRC) on the bits of a number. CRCs are used to detect errors in the transmission of data in digital systems. When a piece of data is sent, a short *check value* is attached to it. The check value is obtained by polynomial division with the bits in the data. When the data is received, the polynomial division is repeated, and the result is compared with the check value. If the results differ, then the data was corrupted during transmission.

Calculate Check Value by Hand

Start with a 16-bit binary number, which is the message to be transmitted:

1101100111011010

To obtain the check value, divide this number by the polynomial $x^3 + x^2 + x + 1$. You can represent this polynomial with its coefficients: 1111.

The division is performed in steps, and after each step the polynomial divisor is aligned with the left-most 1 in the number. Because the result of dividing by the four term polynomial has three bits (in general dividing by a polynomial of length $n + 1$ produces a check value of length n), append the number with 000 to calculate the remainder. At each step, the result uses the bit-wise XOR of the four bits being operated on, and all other bits are unchanged.

The first division is

1101100111011010	000
1111	

0010100111011010	000

Each successive division operates on the result of the previous step, so the second division is

0010100111011010	000
1111	

0001010111011010	000

The division is completed once the dividend is all zeros. The complete division, including the above two steps, is

1101100111011010	000
1111	
0010100111011010	000
1111	
0001010111011010	000
1111	
0000101111011010	000
1111	
0000010011011010	000
1111	
0000001101011010	000
1111	
0000000010011010	000
1111	
0000000001101010	000
1111	

```

0000000000010010 000
      1111
0000000000001100 000
      1111
0000000000000011 000
      11 11
0000000000000000 110

```

The remainder bits, 110, are the check value for this message.

Calculate Check Value Programmatically

In MATLAB®, you can perform this same operation to obtain the check value using bit-wise operations. First, define variables for the message and polynomial divisor. Use unsigned 32-bit integers so that extra bits are available for the remainder.

```

message = 0b1101100111011010u32;
messageLength = 16;
divisor = 0b1111u32;
divisorDegree = 3;

```

Next, initialize the polynomial divisor. Use `dec2bin` to display the bits of the result.

```

divisor = bitshift(divisor,messageLength-divisorDegree-1);
dec2bin(divisor)

ans =
'1111000000000000'

```

Now, shift the divisor and message so that they have the correct number of bits (16 bits for the message and 3 bits for the remainder).

```

divisor = bitshift(divisor,divisorDegree);
remainder = bitshift(message,divisorDegree);
dec2bin(divisor)

ans =
'1111000000000000'

dec2bin(remainder)

ans =
'1101100111011010000'

```

Perform the division steps of the CRC using a `for` loop. The `for` loop always advances a single bit each step, so include a check to see if the current digit is a 1. If the current digit is a 1, then the division step is performed; otherwise, the loop advances a bit and continues.

```

for k = 1:messageLength
    if bitget(remainder,messageLength+divisorDegree)
        remainder = bitxor(remainder,divisor);
    end
    remainder = bitshift(remainder,1);
end

```

Shift the bits of the remainder to the right to get the check value for the operation.

```

CRC_check_value = bitshift(remainder,-messageLength);
dec2bin(CRC_check_value)

```

```
ans =  
'110'
```

Check Message Integrity

You can use the check value to verify the integrity of a message by repeating the same division operation. However, instead of using a remainder of 000 to start, use the check value 110. If the message is error free, then the result of the division will be zero.

Reset the remainder variable, and add the CRC check value to the remainder bits using a bit-wise OR. Introduce an error into the message by flipping one of the bit values with `bitset`.

```
remainder = bitshift(message,divisorDegree);  
remainder = bitor(remainder,CRC_check_value);  
remainder = bitset(remainder,6);  
dec2bin(remainder)  
  
ans =  
'11011100111011110110'
```

Perform the CRC division operation and then check if the result is zero.

```
for k = 1:messageLength  
    if bitget(remainder,messageLength+divisorDegree)  
        remainder = bitxor(remainder,divisor);  
    end  
    remainder = bitshift(remainder,1);  
end  
if remainder == 0  
    disp('Message is error free.')  
else  
    disp('Message contains errors.')  
end
```

Message contains errors.

References

- [1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [2] Wicker, Stephen B. *Error Control Systems for Digital Communication and Storage*. Upper Saddle River, NJ: Prentice Hall, 1995.

See Also

[bitshift](#) | [bitxor](#)

More About

- “Bit-Wise Operations” on page 2-38
- “Hexadecimal and Binary Values” on page 6-55

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 2-51](#)

["Steps for Building Expressions" on page 2-52](#)

["Operators and Characters" on page 2-55](#)

This topic describes what regular expressions are and how to use them to search text. Regular expressions are flexible and powerful, though they use complex syntax. An alternative to regular expressions is a *pattern* (since R2020b), which is simpler to define and results in code that is easier to read. For more information, see "Build Pattern Expressions" on page 6-40.

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.

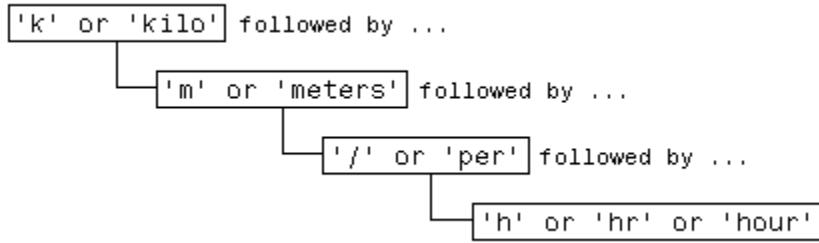
For example, several ways of expressing a metric rate of speed are:

`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(i|o)?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 =
1x2 cell array
{'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
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexp替換	Replace part of text using regular expression.
regexptranslate	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 `regexp替換`, 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 2-53

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 2-53

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 2-54

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

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 <code>jan_stephens</code> ...	One or more lowercase letters and underscores
Add '@' <code>jan_stephens@</code> ...	@ sign
Add the ISP <code>jan_stephens@horizon</code> ...	One or more lowercase letters, no underscores
Add a dot (period) <code>jan_stephens@horizon.</code> ...	Dot (period) character
Finish with the domain <code>jan_stephens@horizon.net</code>	<code>.com</code> or <code>.net</code>

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 =
1x1 cell array
{'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 =
```

```
1x1 cell array
{'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 2-55
- “Character Representation” on page 2-56
- “Quantifiers” on page 2-56
- “Grouping Operators” on page 2-57
- “Anchors” on page 2-58
- “Lookaround Assertions” on page 2-58
- “Logical and Conditional Operators” on page 2-59
- “Token Operators” on page 2-60
- “Dynamic Expressions” on page 2-60
- “Comments” on page 2-61
- “Search Flags” on page 2-61

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'.
[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.

Metacharacter	Description	Example
\w	Any alphabetic, numeric, or underscore character. For English character sets, \w is equivalent to [a-zA-Z_0-9]	'\w*' identifies a word comprised of any grouping of alphabetic, numeric, or underscore characters.
\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 comprised of any grouping of alphabetic, numeric, or underscore characters.
\s	Any white-space character; equivalent to [\f\n\r\t\v]	'\w*\n\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.
\o{N} or \o{N}	Character of octal value N	'\o{40}' matches the space character, defined by octal 40.
\x{N} 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
\v	Vertical tab
\char	Any character with special meaning in regular expressions that you want to match literally (for example, use \\ to match a single backslash)

Quantifiers

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

Quantifier	Number of Times Expression Occurs	Example
expr*	0 or more times consecutively.	'\w*' matches a word of any length.
expr?	0 times or 1 time.	'\w*(\.\m)?' matches words that optionally end with the extension .m.

Quantifier	Number of Times Expression Occurs	Example
expr ⁺	1 or more times consecutively.	'' matches an HTML tag when the file name contains one or more characters.
expr{m,n}	At least m times, but no more than n times consecutively. {0,1} is equivalent to ?.	'\S{4,8}' matches between four and eight non-white-space characters.
expr{m,}	At least m times consecutively. {0,} and {1,} are equivalent to * and +, respectively.	'' matches an <a> HTML tag when the file name contains one or more characters.
expr{n}	Exactly n times consecutively. Equivalent to {n,n}.	'\d{4}' 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
expr ^q	Greedy expression: match as many characters as possible.	Given the text '<tr><td><p>text</p></td>', the expression '</?t.*>' matches all characters between <tr> and /td>: '<tr><td><p>text</p></td>'
expr ^{q?}	Lazy expression: match as few characters as necessary.	Given the text '<tr><td><p>text</p></td>', the expression '</?t.*?>' ends each match at the first occurrence of the closing angle bracket (>): '<tr>' '<td>' '</td>'
expr ^{q+}	Possessive expression: match as much as possible, but do not rescan any portions of the text.	Given the text '<tr><td><p>text</p></td>', the expression '</?t.*+>' does not return any matches, because the closing angle bracket is captured using .*, 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
(expr)	Group elements of the expression and capture tokens.	'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.

Grouping Operator	Description	Example
(?:expr)	Group, but do not capture tokens.	'(?:[aeiou][^aeiou])\{2\}' matches two consecutive patterns of a vowel followed by a nonvowel, such as 'anon'. Without grouping, '[aeiou][^aeiou]\{2\}' matches a vowel followed by two nonvowels.
(?>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> .	'\w*(?=ing)' matches terms that are followed by <code>ing</code> , such as 'Fly' and 'fall' in the input text 'Flying, not falling.'
<code>expr(?!test)</code>	Look ahead for characters that do not match <code>test</code> .	'i(?!ng)' 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> .	'(?<=re)\w*' matches terms that follow ' <code>re</code> ', such as 'new', 'use', and 'cycle' in the input text 'renew, reuse, recycle'
<code>(?<!test)expr</code>	Look behind for characters that do not match <code>test</code> .	'(?<!\\d)(\\d)(?!\\d)' 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> .	'(?=[a-z])[^aeiou]' matches consonants.
<code>(?!test)expr</code>	Match <code>expr</code> and do not match <code>test</code> .	'(?! [aeiou])[a-z]' matches consonants.

For more information, see "Lookahead Assertions in Regular Expressions" on page 2-63.

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 2-58.)

Conditions can be tokens on page 2-60, lookahead assertions on page 2-58, or dynamic expressions on page 2-60 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.	'(let tel)\\w+' matches words that start with <code>let</code> or <code>tel</code> .
<code>(?(cond)expr)</code>	If condition <code>cond</code> is <code>true</code> , then match <code>expr</code> .	'(?(@ispc)[A-Z]:\\ \\)' 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 <code>true</code> , then match <code>expr1</code> . Otherwise, match <code>expr2</code> .	'Mr(s?)\\..*?(?(1)her his) \\w*' 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
(expr)	Capture in a token the characters that match the enclosed expression.	'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.
\N	Match the Nth token.	'<(\w+).*>.*</\1>' captures tokens for HTML tags, such as 'title' from the text '<title>Some text</title>'.
(?(N)expr1 expr2)	If the Nth token is found, then match expr1. Otherwise, match expr2.	'Mr(s?)\.*?(?(1)her his) \w*' matches text that includes her when the text begins with Mrs, or that includes his when the text begins with Mr.

Named Token Operator	Description	Example
(?<name>expr)	Capture in a named token the characters that match the enclosed expression.	'(?<month>\d+) - (?<day>\d+) - (?<year>\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 2-66.

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)	<p>Parse <code>expr</code> and include the resulting term in the match expression.</p> <p>When parsed, <code>expr</code> must correspond to a complete, valid regular expression.</p> <p>Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of <code>expr</code>, and one for the complete match.</p>	'^(\\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 <code>cmd</code> , 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 <code>cmd</code> , 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, <code>cmd</code>

For more information, see “Dynamic Regular Expressions” on page 2-72.

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

Search Flags

Search flags modify the behavior for matching expressions.

Flag	Description
(?-i)	Match letter case (default for <code>regexp</code> and <code>regexprep</code>).
(?i)	Do not match letter case (default for <code>regexpi</code>).

Flag	Description
(?s)	Match dot (.) in the pattern with any character (default).
(?-s)	Match dot in the pattern with any character that is not a newline character.
(?-m)	Match the ^ and \$ metacharacters at the beginning and end of text (default).
(?m)	Match the ^ and \$ metacharacters at the beginning and end of a line.
(?-x)	Include space characters and comments when matching (default).
(?x)	Ignore space characters and comments when matching. Use '\ ' and '\#' to match space and # 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

`pattern` | `regexp` | `regexpi` | `regexprep` | `regexptranslate`

More About

- “Lookahead Assertions in Regular Expressions” on page 2-63
- “Tokens in Regular Expressions” on page 2-66
- “Dynamic Regular Expressions” on page 2-72
- “Search and Replace Text” on page 6-37

Lookahead Assertions in Regular Expressions

In this section...

- ["Lookahead Assertions" on page 2-63](#)
- ["Overlapping Matches" on page 2-63](#)
- ["Logical AND Conditions" on page 2-64](#)

Lookahead Assertions

There are two types of lookaround 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 =
1×1 cell array
{'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 =
```



```
1x43 cell array
```



```
Columns 1 through 14
```



```
{' '}  {'N'}  {'0'}  {'R'}  {'M'}  {'E'}  {'S'}  {'T'}  {' '}  {'E'}  {'S'}
```



```
Columns 15 through 28
```



```
{' '}  {'t'}  {'h'}  {' '}  {'m'}  {'t'}  {'r'}  {'x'}  {' '}  {'2'}  {'S'}
```



```
Columns 29 through 42
```



```
{'.'}  {'e'}  {' '}  {' '}  {' '}  {' '}  {'N'}  {'O'}  {'R'}  {'M'}  {'S'}
```



```
Column 43
```



```
{'S'}
```

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 =
```

```
1x13 cell array  
{'s'} {'t'} {'m'} {'t'} {'t'} {'h'} {'m'} {'t'} {'r'} {'x'} {'t'} {'t'}
```

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` | `regexpi` | `regexprep`

More About

- “Regular Expressions” on page 2-51

Tokens in Regular Expressions

In this section...

- “Introduction” on page 2-66
- “Multiple Tokens” on page 2-68
- “Unmatched Tokens” on page 2-69
- “Tokens in Replacement Text” on page 2-69
- “Named Capture” on page 2-70

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 =

1x4 cell array

{'dd'}     {'pp'}     {'dd'}     {'pp'}
```

The cell array `tok` contains cell arrays that each contain a token.

```
tok{::}

ans =

1x1 cell array

{'d'}
```

ans =

```
1x1 cell array
```

```
{'p'}
```

```
ans =
```

```
1x1 cell array
```

```
{'d'}
```

```
ans =
```

```
1x1 cell array
```

```
{'p'}
```

The cell array `ext` contains numeric arrays that each contain starting and ending indices for a token.

```
ext{::}
```

```
ans =
```

```
11     11
```

```
ans =
```

```
26     26
```

```
ans =
```

```
35     35
```

```
ans =
```

```
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 =

1x1 cell array

{'a'}

ans =

1x1 cell array

{'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\bpascal\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 =
1x6 cell array
{'C:'}    {'WINNT'}    {0x0 char}    {'bpascal'}    {'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.

```
regexp('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 =  
1x4 cell array  
{'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 =  
struct with fields:  
  
adrs: '134 Main Street'  
city: 'Boulder'  
state: 'CO'  
zip: '14923'
```

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

```
loc2 =
```

struct with fields:

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

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

```
loc3 =
```

struct with fields:

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

See Also

[regexp](#) | [regexpi](#) | [regexprep](#)

More About

- “Regular Expressions” on page 2-51

Dynamic Regular Expressions

In this section...

- “Introduction” on page 2-72
- “Dynamic Match Expressions — (??expr)” on page 2-73
- “Commands That Modify the Match Expression — (??@cmd)” on page 2-73
- “Commands That Serve a Functional Purpose — (@cmd)” on page 2-74
- “Commands in Replacement Expressions — \${cmd}” on page 2-76

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'

regexp('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', '8XXXXXXXXX', '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 =
1x3 cell array

{'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 = regexp替换成(chr, '\W*', '')  
  
chr =  
  
    'findthepalindromeneveroddoreveninthisstring'
```

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

```
palindrome = regexp(chr, '(.{3,}).?(??@fliplr($1))', 'match')  
  
palindrome =  
  
1x1 cell array  
  
{'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;  
  
palindrome = regexp(chr, '(.{3,}).?(??@fun($1))', 'match')  
  
palindrome =  
  
1x1 cell array  
  
{'neveroddoreven'}
```

Commands That Serve a Functional Purpose – (?@cmd)

The `(?@cmd)` operator specifies a MATLAB command that `regexp` or `regexp替换成` 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 =  
  
1x1 cell array
```

```
{'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 =
1x1 cell array

{'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 =
1x1 cell array

{'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 =
1x1 cell array

{'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 =
1x6 cell array
{'Euler Cauchy Bo...' }    {'Euler Cauchy ' }    {'Euler ' }    {'Cauchy Boole' }    {'Cauchy ' }
```

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

regexp(chr, expr, 'once');

starting match: [^abcdefghijkl]
starting match: [a^abcdefghijkl]
starting match: [ab^abcdefghijkl]
starting match: [abc^abcdefghijkl]
starting match: [abcd^abcdefghijkl]
starting match: [abcde^abcdefghijkl]
starting match: [abcdef^abcdefghijkl]
```

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

```
regexp替('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 `regexp`, passing in values for the quantity to be converted and name of the imperial unit:

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

ans =
'This highway is 201.1625 kilometers long'

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

ans =
'This pitcher holds 1.1828 litres of water'

regexp('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 `regexp` command uses the array `A` defined in the base workspace:

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

```
A =  
8     1     6  
3     5     7  
4     9     2  
  
regexp替('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](#) | [regexp替](#)

More About

- “Regular Expressions” on page 2-51

Comma-Separated Lists

In this section...

- “What Is a Comma-Separated List?” on page 2-79
- “Generating a Comma-Separated List” on page 2-79
- “Assigning Output from a Comma-Separated List” on page 2-81
- “Assigning to a Comma-Separated List” on page 2-81
- “How to Use the Comma-Separated Lists” on page 2-82
- “Fast Fourier Transform Example” on page 2-84

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 =

```

```
[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 =
```

```
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 2-83
- “Displaying Arrays” on page 2-83

- “Concatenation” on page 2-83
- “Function Call Arguments” on page 2-84
- “Function Return Values” on page 2-84

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

```

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 2-86
- “Variables with Sequential Names” on page 2-86
- “Files with Sequential Names” on page 2-87
- “Function Names in Variables” on page 2-87
- “Field Names in Variables” on page 2-88
- “Error Handling” on page 2-88

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
- “Structure Arrays” 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:

- “Choose Command Syntax or Function Syntax” on page 1-6
- “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 11-10.

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)

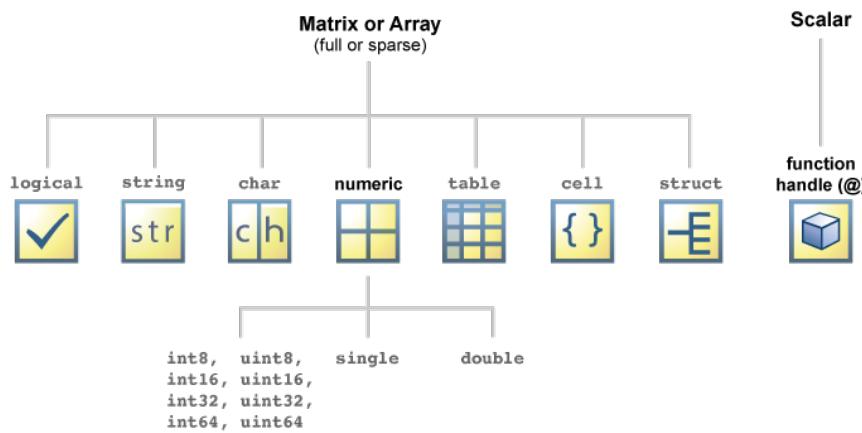
Overview of MATLAB Classes

Fundamental MATLAB Classes

There are many different data types, or classes, that you can work with in MATLAB. You can build matrices and arrays of floating-point and integer data, characters and strings, logical `true` and `false` values, and so on. Function handles connect your code with any MATLAB function regardless of the current scope. Tables, timetables, 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 4-6	<ul style="list-style-type: none"> Required for fractional numeric data. Double on page 4-6 and Single on page 4-6 precision. Use <code>realmin</code> and <code>realmax</code> to show range of values on page 4-9. 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 30-2 Use <code>intmin</code> and <code>intmax</code> to show range of values on page 4-4. 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 2-51. 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>.
<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> , <code>timetable</code>	“Tables”, “Timetables”	<ul style="list-style-type: none"> Tables are rectangular containers for mixed-type, column-oriented data. Tables have row and variable names that identify contents. Timetables also provide storage for data in a table with rows labeled by time. Timetable functions can synchronize, resample, or aggregate timestamped data. Use the properties of a table or timetable 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.

Class Name	Documentation	Intended Use
struct	"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 2-79.More memory required for overhead
cell	"Cell Arrays"	<ul style="list-style-type: none">Cells store arrays of varying classes and sizes.Allows freedom to package data as you want.Manipulation of elements is similar to 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 4-6
- “Create Complex Numbers” on page 4-13
- “Infinity and NaN” on page 4-14
- “Identifying Numeric Classes” on page 4-16
- “Display Format for Numeric Values” on page 4-17
- “Integer Arithmetic” on page 4-19
- “Single Precision Math” on page 4-26

Integers

In this section...

- “Integer Classes” on page 4-2
- “Creating Integer Data” on page 4-2
- “Arithmetic Operations on Integer Classes” on page 4-4
- “Largest and Smallest Values for Integer Classes” on page 4-4

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	<code>int8</code>
Signed 16-bit integer	- 2^{15} to 2^{15} -1	<code>int16</code>
Signed 32-bit integer	- 2^{31} to 2^{31} -1	<code>int32</code>
Signed 64-bit integer	- 2^{63} to 2^{63} -1	<code>int64</code>
Unsigned 8-bit integer	0 to 2^8 -1	<code>uint8</code>
Unsigned 16-bit integer	0 to 2^{16} -1	<code>uint16</code>
Unsigned 32-bit integer	0 to 2^{32} -1	<code>uint32</code>
Unsigned 64-bit integer	0 to 2^{64} -1	<code>uint64</code>

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 =
1x11 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.

Operations involving complex numbers with integer types is not supported.

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 4-6
- “Single-Precision Floating Point” on page 4-6
- “Creating Floating-Point Data” on page 4-6
- “Arithmetic Operations on Floating-Point Numbers” on page 4-8
- “Largest and Smallest Values for Floating-Point Classes” on page 4-9
- “Accuracy of Floating-Point Data” on page 4-10
- “Avoiding Common Problems with Floating-Point Arithmetic” on page 4-11

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)
30 to 23	Exponent, biased by 127
22 to 0	Fraction f of the number 1.f

Because MATLAB stores numbers of type `single` using 32 bits, they require less memory than numbers of type `double`, which use 64 bits. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`.

Creating Floating-Point Data

Use double-precision to store values greater than approximately 3.4×10^{38} or less than approximately -3.4×10^{38} . For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Creating Double-Precision Data

Because the default numeric type for MATLAB is `double`, you can create a `double` with a simple assignment statement:

```
x = 25.783;
```

The `whos` function shows that MATLAB has created a 1-by-1 array of type `double` for the value you just stored in `x`:

whos x			
Name	Size	Bytes	Class
x	1x1	8	double

Use `isfloat` if you just want to verify that `x` is a floating-point number. This function returns logical 1 (`true`) if the input is a floating-point number, and logical 0 (`false`) otherwise:

```
isfloat(x)
ans =
```

```
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 $5 + \text{eps}(5)$. 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 `pi` 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 `expml` 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 $\mathbf{Ax} = \mathbf{b}$ 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.

Create 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 4-14](#)

["NaN" on page 4-14](#)

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 = 1.e1000</code>
<code>x =</code>	<code>x =</code>
<code>Inf</code>	<code>Inf</code>
<code>x = exp(1000)</code>	<code>x = log(0)</code>
<code>x =</code>	<code>x =</code>
<code>Inf</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 =
ffff80000000000000000000000000
```

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 x .
<code>xType = class(x);</code>	Assign the data type of x to a variable.
<code>isnumeric(x)</code>	Determine if x 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 x 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 x is real or complex.
<code>isnan(x)</code>	Determine if x is Not a Number (NaN).
<code>isinf(x)</code>	Determine if x is infinite.
<code>isfinite(x)</code>	Determine if x is finite.

Display Format for Numeric Values

By default, MATLAB uses a 5-digit short format to display numbers. For example,

```
x = 4/3
x =
1.3333
```

You can change the display in the Command Window or Editor using the `format` function.

```
format long
x
x =
1.333333333333333
```

Using the `format` function only sets the format for the current MATLAB session. To set the format for subsequent sessions, click  **Preferences** on the **Home** tab in the **Environment** section. Select **MATLAB > Command Window**, and then choose a **Numeric format** option.

The following table summarizes the numeric output format options.

Style	Result	Example
<code>short</code> (default)	Short, fixed-decimal format with 4 digits after the decimal point.	3.1416
<code>long</code>	Long, fixed-decimal format with 15 digits after the decimal point for <code>double</code> values, and 7 digits after the decimal point for <code>single</code> values.	3.141592653589793
<code>shortE</code>	Short scientific notation with 4 digits after the decimal point.	3.1416e+00
<code>longE</code>	Long scientific notation with 15 digits after the decimal point for <code>double</code> values, and 7 digits after the decimal point for <code>single</code> values.	3.141592653589793e+00
<code>shortG</code>	Short, fixed-decimal format or scientific notation, whichever is more compact, with a total of 5 digits.	3.1416
<code>longG</code>	Long, fixed-decimal format or scientific notation, whichever is more compact, with a total of 15 digits for <code>double</code> values, and 7 digits for <code>single</code> values.	3.14159265358979
<code>shortEng</code>	Short engineering notation (exponent is a multiple of 3) with 4 digits after the decimal point.	3.1416e+000
<code>longEng</code>	Long engineering notation (exponent is a multiple of 3) with 15 significant digits.	3.14159265358979e+000
<code>+</code>	Positive/Negative format with +, -, and blank characters displayed for positive, negative, and zero elements.	+

Style	Result	Example
bank	Currency format with 2 digits after the decimal point.	3.14
hex	Hexadecimal representation of a binary double-precision number.	400921fb54442d18
rat	Ratio of small integers.	355/113

The display format only affects how numbers are displayed, not how they are stored in MATLAB.

See Also

`format`

Related Examples

- “Format Output”

Integer Arithmetic

This example shows how to perform arithmetic on integer data representing signals and images.

Load Integer Signal Data

Load measurement datasets comprising signals from four instruments using 8 and 16-bit A-to-D's resulting in data saved as `int8`, `int16` and `uint16`. Time is stored as `uint16`.

```
load integersignal

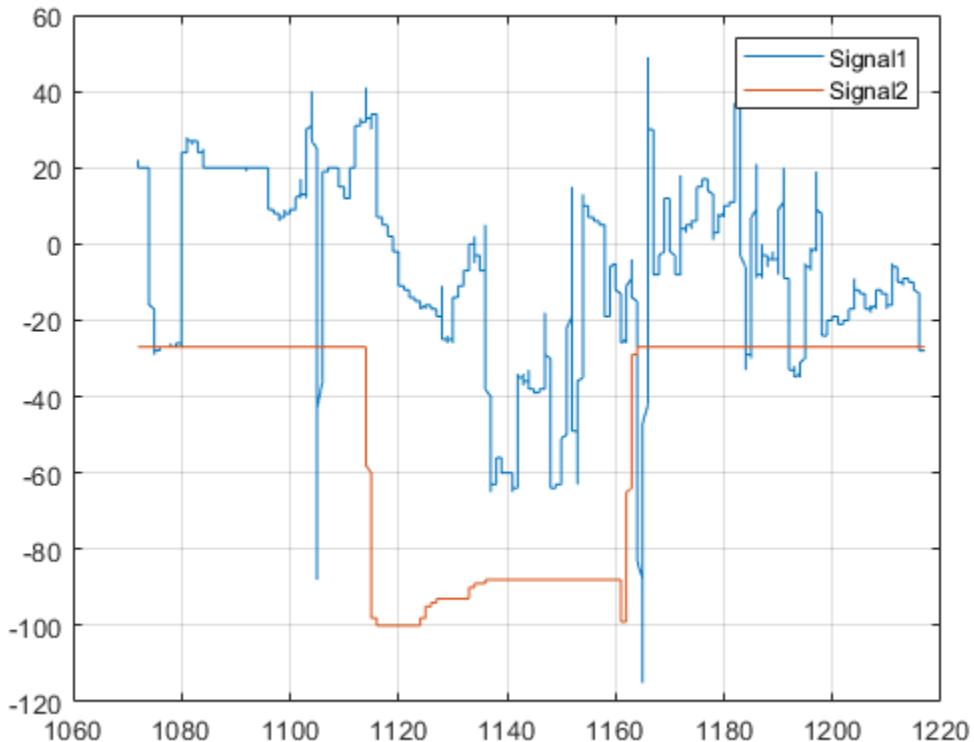
% Look at variables
whos Signal1 Signal2 Signal3 Signal4 Time1
```

Name	Size	Bytes	Class	Attributes
Signal1	7550x1	7550	int8	
Signal2	7550x1	7550	int8	
Signal3	7550x1	15100	int16	
Signal4	7550x1	15100	uint16	
Time1	7550x1	15100	uint16	

Plot Data

First we will plot two of the signals to see the signal ranges.

```
plot(Time1, Signal1, Time1, Signal2);
grid;
legend('Signal1','Signal2');
```



It is likely that these values would need to be scaled to calculate the actual physical value that the signal represents e.g. Volts.

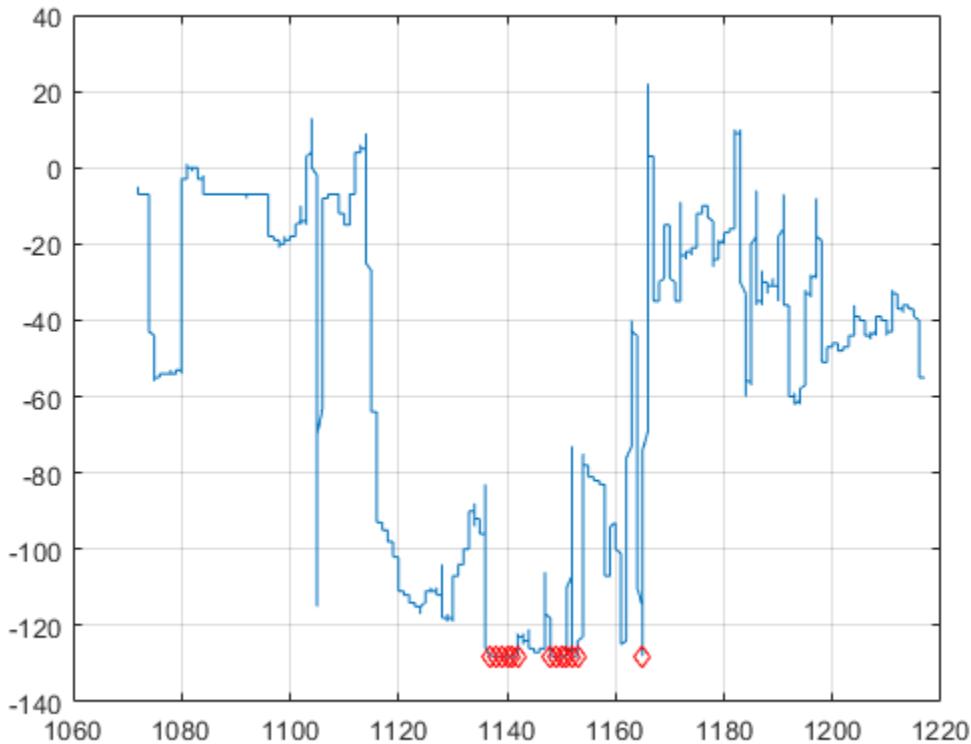
Process Data

We can perform standard arithmetic on integers such as +, -, *, and /. Let's say we wished to find the sum of Signal1 and Signal2.

```
SumSig = Signal1 + Signal2; % Here we sum the integer signals.
```

Now let's plot the sum signal and see where it saturates.

```
cla;
plot(Time1, SumSig);
hold on
Saturated = (SumSig == intmin('int8')) | (SumSig == intmax('int8')); % Find where it has saturated
plot(Time1(Saturated),SumSig(Saturated), 'rd')
grid
hold off
```



The markers show where the signal has saturated.

Load Integer Image Data

Next we will look at arithmetic on some image data.

```
street1 = imread('street1.jpg'); % Load image data
street2 = imread('street2.jpg');
whos street1 street2
```

Name	Size	Bytes	Class	Attributes
street1	480x640x3	921600	uint8	
street2	480x640x3	921600	uint8	

Here we see the images are 24-bit color, stored as three planes of `uint8` data.

Display Images

Display first image.

```
cla;
image(street1); % Display image
axis equal
axis off
```



Display second image

```
image(street2); % Display image  
axis equal  
axis off
```



Scale an Image

We can scale the image by a double precision constant but keep the image stored as integers. For example,

```
duller = 0.5 * street2; % Scale image with a double constant but create an integer
whos duller
```

Name	Size	Bytes	Class	Attributes
duller	480x640x3	921600	uint8	

```
subplot(1,2,1);
image(street2);
axis off equal tight
title('Original'); % Display image

subplot(1,2,2);
image(duller);
axis off equal tight
title('Duller'); % Display image
```



Add the Images

We can add the two street images together and plot the ghostly result.

```
combined = street1 + duller; % Add |uint8| images
subplot(1,1,1)
cla;
image(combined); % Display image
title('Combined');
axis equal
axis off
```

Combined



Single Precision Math

This example shows how to perform arithmetic and linear algebra with single precision data. It also shows how the results are computed appropriately in single-precision or double-precision, depending on the input.

Create Double Precision Data

Let's first create some data, which is double precision by default.

```
Ad = [1 2 0; 2 5 -1; 4 10 -1]
```

```
Ad = 3x3
```

1	2	0
2	5	-1
4	10	-1

Convert to Single Precision

We can convert data to single precision with the `single` function.

```
A = single(Ad); % or A = cast(Ad,'single');
```

Create Single Precision Zeros and Ones

We can also create single precision zeros and ones with their respective functions.

```
n = 1000;
Z = zeros(n,1,'single');
O = ones(n,1,'single');
```

Let's look at the variables in the workspace.

```
whos A Ad O Z n
```

Name	Size	Bytes	Class	Attributes
A	3x3	36	single	
Ad	3x3	72	double	
O	1000x1	4000	single	
Z	1000x1	4000	single	
n	1x1	8	double	

We can see that some of the variables are of type `single` and that the variable A (the single precision version of Ad) takes half the number of bytes of memory to store because singles require just four bytes (32-bits), whereas doubles require 8 bytes (64-bits).

Arithmetic and Linear Algebra

We can perform standard arithmetic and linear algebra on singles.

```
B = A'      % Matrix Transpose
```

```
B = 3x3 single matrix
```

1	2	4
---	---	---

2	5	10
0	-1	-1

`whos B`

Name	Size	Bytes	Class	Attributes
B	3x3	36	single	

We see the result of this operation, B, is a single.

`C = A * B % Matrix multiplication`

`C = 3x3 single matrix`

5	12	24
12	30	59
24	59	117

`C = A .* B % Elementwise arithmetic`

`C = 3x3 single matrix`

1	4	0
4	25	-10
0	-10	1

`X = inv(A) % Matrix inverse`

`X = 3x3 single matrix`

5	2	-2
-2	-1	1
0	-2	1

`I = inv(A) * A % Confirm result is identity matrix`

`I = 3x3 single matrix`

1	0	0
0	1	0
0	0	1

`I = A \ A % Better way to do matrix division than inv`

`I = 3x3 single matrix`

1	0	0
0	1	0
0	0	1

`E = eig(A) % Eigenvalues`

`E = 3x1 single column vector`

```
3.7321
0.2679
1.0000

F = fft(A(:,1)) % FFT
F = 3x1 single column vector
7.0000 + 0.0000i
-2.0000 + 1.7321i
-2.0000 - 1.7321i

S = svd(A) % Singular value decomposition
S = 3x1 single column vector
12.3171
0.5149
0.1577

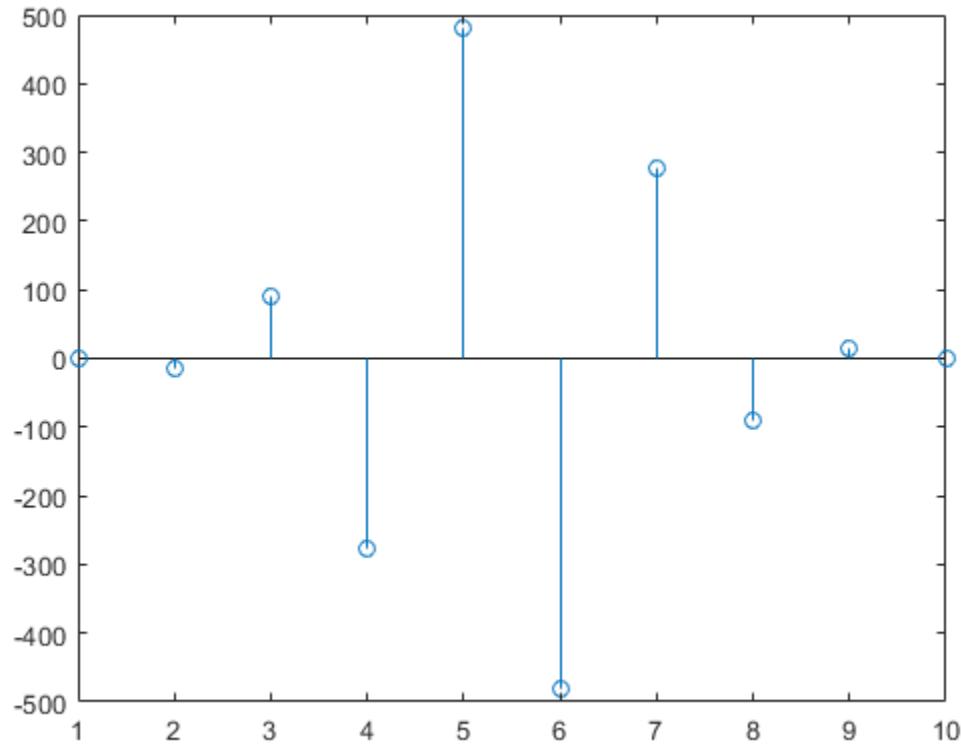
P = round(poly(A)) % The characteristic polynomial of a matrix
P = 1x4 single row vector
1      -5      5      -1

R = roots(P) % Roots of a polynomial
R = 3x1 single column vector
3.7321
1.0000
0.2679

Q = conv(P,P) % Convolve two vectors
Q = 1x7 single row vector
1     -10     35    -52     35    -10      1

R = conv(P,Q)
R = 1x10 single row vector
1     -15     90   -278    480   -480    278   -90     15     -1

stem(R); % Plot the result
```



A Program that Works for Either Single or Double Precision

Now let's look at a function to compute enough terms in the Fibonacci sequence so the ratio is less than the correct machine epsilon (`eps`) for datatype single or double.

```
% How many terms needed to get single precision results?
fibodemo('single')
```

```
ans = 19
```

```
% How many terms needed to get double precision results?
fibodemo('double')
```

```
ans = 41
```

```
% Now let's look at the working code.
type fibodemo
```

```
function nterms = fibodemo(dtype)
%FIBODEMO Used by SINGLEMATH demo.
% Calculate number of terms in Fibonacci sequence.
```

```
% Copyright 1984-2014 The MathWorks, Inc.
```

```
fcurrent = ones(dtype);
fnext = fcurrent;
```

```
goldenMean = (ones(dtype)+sqrt(5))/2;
tol = eps(goldenMean);
nterms = 2;
while abs(fnext/fcurrent - goldenMean) >= tol
    nterms = nterms + 1;
    temp = fnext;
    fnext = fnext + fcurrent;
    fcurrent = temp;
end
```

Notice that we initialize several of our variables, `fcurrent`, `fnext`, and `goldenMean`, with values that are dependent on the input datatype, and the tolerance `tol` depends on that type as well. Single precision requires that we calculate fewer terms than the equivalent double precision calculation.

The Logical Class

- “Find Array Elements That Meet a Condition” on page 5-2
- “Reduce Logical Arrays to Single Value” on page 5-6

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 ($>$, $<$, \geq , \leq , \neq , \approx) 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 = 5×5

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 = 8x1
```

```
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 = 5x1
```

```
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 | A<9).

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

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

```
ans = 3x1
```

```
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 `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 = 5x5
```

```
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 = 5x5
```

```
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

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`

Characters and Strings

- “Text in String and Character Arrays” on page 6-2
- “Create String Arrays” on page 6-5
- “Cell Arrays of Character Vectors” on page 6-12
- “Analyze Text Data with String Arrays” on page 6-15
- “Test for Empty Strings and Missing Values” on page 6-20
- “Formatting Text” on page 6-24
- “Compare Text” on page 6-32
- “Search and Replace Text” on page 6-37
- “Build Pattern Expressions” on page 6-40
- “Convert Numeric Values to Text” on page 6-45
- “Convert Text to Numeric Values” on page 6-49
- “Unicode and ASCII Values” on page 6-53
- “Hexadecimal and Binary Values” on page 6-55
- “Frequently Asked Questions About String Arrays” on page 6-59
- “Update Your Code to Accept Strings” on page 6-64
- “Function Summary” on page 6-72

Text in String and Character Arrays

There are two ways to represent text in MATLAB®. Starting in R2016b, you can store text in *string arrays*. And in any version of MATLAB, you can store text in character arrays. A typical use for character arrays is to store pieces of text as *character vectors*. MATLAB displays strings with double quotes and character vectors with single quotes.

Represent Text with String Arrays

You can store any 1-by-n sequence of characters as a string, using the `string` data type. Starting in R2017a, enclose text in double quotes to create a string.

```
str = "Hello, world"  
  
str =  
"Hello, world"
```

Though the text "Hello, world" is 12 characters long, `str` itself is a 1-by-1 string, or *string scalar*. You can use a string scalar to specify a file name, plot label, or any other piece of textual information.

To find the number of characters in a string, use the `strlength` function.

```
n = strlength(str)  
  
n = 12
```

If the text includes double quotes, use two double quotes within the definition.

```
str = "They said, ""Welcome!"" and waved."  
  
str =  
"They said, "Welcome!" and waved."
```

To add text to the end of a string, use the plus operator, `+`. If a variable can be converted to a string, then `plus` converts it and appends it.

```
fahrenheit = 71;  
celsius = (fahrenheit-32)/1.8;  
tempText = "temperature is " + celsius + "C"  
  
tempText =  
"temperature is 21.6667C"
```

Starting in R2019a, you can also concatenate text using the `append` function.

```
tempText2 = append("Today's ", tempText)  
  
tempText2 =  
"Today's temperature is 21.6667C"
```

The `string` function can convert different types of inputs, such as numeric, datetime, duration, and categorical values. For example, convert the output of `pi` to a string.

```
ps = string(pi)  
  
ps =  
"3.1416"
```

You can store multiple pieces of text in a string array. Each element of the array can contain a string having a different number of characters, without padding.

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

str = 2x3 string
    "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.

```
N = strlength(str)

N = 2×3

    7      6      6
    6      8      3
```

As of R2018b, string arrays are supported throughout MATLAB and MathWorks® products. Functions that accept character arrays (and cell arrays of character vectors) as inputs also accept string arrays.

Represent Text with Character Vectors

To store a 1-by-n sequence of characters as a character vector, using the `char` data type, enclose it in single quotes.

```
chr = 'Hello, world'

chr =
'Hello, world'
```

The text '`Hello, world`' is 12 characters long, and `chr` stores it as a 1-by-12 character vector.

```
whos chr

  Name      Size            Bytes  Class      Attributes
  chr        1x12           24    char
```

If the text includes single quotes, use two single quotes within the definition.

```
chr = 'They said, ''Welcome!'' and waved.'

chr =
'They said, 'Welcome!' and waved.'
```

Character vectors have two principal uses:

- To specify single pieces of text, such as file names and plot labels.
- To represent data that is encoded using characters. In such cases, you might need easy access to individual characters.

For example, you can store a DNA sequence as a character vector.

```
seq = 'GCTAGAATCC';
```

You can access individual characters or subsets of characters by indexing, just as you would index into a numeric array.

```
seq(4:6)
```

```
ans =  
'AGA'
```

Concatenate character vector with square brackets, just as you concatenate other types of arrays.

```
seq2 = [seq 'ATTAGAAACC']
```

```
seq2 =  
'GCTAGAACATTAGAAACC'
```

Starting in R2019a, you also can concatenate text using `append`. The `append` function is recommended because it treats string arrays, character vectors, and cell arrays of character vectors consistently.

```
seq2 = append(seq, 'ATTAGAAACC')
```

```
seq2 =  
'GCTAGAACATTAGAAACC'
```

MATLAB functions that accept string arrays as inputs also accept character vectors and cell arrays of character vectors.

See Also

[append](#) | [cellstr](#) | [char](#) | [horzcat](#) | [plus](#) | [string](#) | [strlength](#)

Related Examples

- “Create String Arrays” on page 6-5
- “Analyze Text Data with String Arrays” on page 6-15
- “Frequently Asked Questions About String Arrays” on page 6-59
- “Update Your Code to Accept Strings” on page 6-64
- “Cell Arrays of Character Vectors” on page 6-12

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 sequence of characters.

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 containing multiple strings 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
    "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
    {'Mercury'}    {'Venus'}    {'Earth'}
```



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

```
str = 1x3 string
    "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
    "5"      "10"     "20"     "3.1416"
```

Convert a datetime value to a string.

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



```
ans =
"24-Feb-2021 01:15:51"
```

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
    ""    ""
    ""    ""
    ""    ""
```

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
    ""      "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 the columns.

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

ans = 1x3 string
    "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
    "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
"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
"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
"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
    "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
    "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
    "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  
  "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  
  "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 6-15
- “Search and Replace Text” on page 6-37
- “Compare Text” on page 6-32
- “Test for Empty Strings and Missing Values” on page 6-20
- “Frequently Asked Questions About String Arrays” on page 6-59
- “Update Your Code to Accept Strings” on page 6-64

Cell Arrays of Character Vectors

To store text as a *character vector*, enclose it single quotes. Typically, a character vector has text that you consider to be a single piece of information, such as a file name or a label for a plot. If you have many pieces of text, such as a list of file names, then you can store them in a cell array. A cell array whose elements are all character vectors is a *cell array of character vectors*.

Note

- As of R2018b, the recommended way to store text is to use *string arrays*. If you create variables that have the `string` data type, store them in string arrays, not cell arrays. For more information, see “Text in String and Character Arrays” on page 6-2 and “Update Your Code to Accept Strings” on page 6-64.
 - While the phrase *cell array of strings* frequently has been used to describe such cell arrays, the phrase is no longer accurate because such a cell array holds character vectors, not strings.
-

Create Cell Array of Character Vectors

To create a cell array of character vectors, use curly braces, `{}`, just as you would to create any cell array. For example, use a cell array of character vectors to store a list of names.

```
C = {'Li', 'Sanchez', 'Jones', 'Yang', 'Larson'}
```

```
C = 1x5 cell
    {'Li'}    {'Sanchez'}    {'Jones'}    {'Yang'}    {'Larson'}
```

The character vectors in `C` can have different lengths because a cell array does not require that its contents have the same size. To determine the lengths of the character vectors in `C`, use the `strlength` function.

```
L = strlength(C)
```

```
L = 1x5
    2      7      5      4      6
```

Access Character Vectors in Cell Array

To access character vectors in a cell array, index into it using curly braces, `{}`. Extract the contents of the first cell and store it as a character vector.

```
C = {'Li', 'Sanchez', 'Jones', 'Yang', 'Larson'};
chr = C{1}

chr =
'Li'
```

Assign a different character vector to the first cell.

```
C{1} = 'Yang'
```

```
C = 1x5 cell
  {'Yang'}    {'Sanchez'}   {'Jones'}   {'Yang'}   {'Larson'}
```

To refer to a subset of cells, instead of their contents, index using smooth parentheses.

```
C(1:3)
```

```
ans = 1x3 cell
  {'Yang'}    {'Sanchez'}   {'Jones'}
```

While you can access the contents of cells by indexing, most functions that accept cell arrays as inputs operate on the entire cell array. For example, you can use the `strcmp` function to compare the contents of C to a character vector. `strcmp` returns 1 where there is a match and 0 otherwise.

```
TF = strcmp(C, 'Yang')
```

```
TF = 1x5 logical array
```

```
1 0 0 1 0
```

You can sum over TF to find the number of matches.

```
num = sum(TF)
```

```
num = 2
```

Use TF as logical indices to return the matches in C. If you index using smooth parentheses, then the output is a cell array containing only the matches.

```
M = C(TF)
```

```
M = 1x2 cell
  {'Yang'}    {'Yang'}
```

Convert Cell Arrays to String Arrays

As of R2018b, string arrays are supported throughout MATLAB® and MathWorks® products. Therefore it is recommended that you use string arrays instead of cell arrays of character vectors. (However, MATLAB functions that accept string arrays as inputs do accept character vectors and cell arrays of character vectors as well.)

You can convert cell arrays of character vectors to string arrays. To convert a cell array of character vectors, use the `string` function.

```
C = {'Li', 'Sanchez', 'Jones', 'Yang', 'Larson'}
```

```
C = 1x5 cell
  {'Li'}    {'Sanchez'}   {'Jones'}   {'Yang'}   {'Larson'}
```

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

```
str = 1x5 string
    "Li"      "Sanchez"     "Jones"      "Yang"      "Larson"
```

In fact, the `string` function converts any cell array, so long as all of the contents can be converted to strings.

```
C2 = {5, 10, 'some text', datetime('today')}
C2=1x4 cell array
    {[5]}    {[10]}    {'some text'}    {[24-Feb-2021]}

str2 = string(C2)
str2 = 1x4 string
    "5"      "10"      "some text"      "24-Feb-2021"
```

See Also

`cellstr` | `char` | `iscellstr` | `strcmp` | `string`

More About

- “Text in String and Character Arrays” on page 6-2
- “Access Data in Cell Array” on page 12-5
- “Create String Arrays” on page 6-5
- “Update Your Code to Accept Strings” on page 6-64
- “Frequently Asked Questions About String Arrays” on page 6-59

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 `splitlevels` 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 = splitlevels(sonnets);
sonnets(1:5)

ans = 5x1 string
"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
"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
"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
"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"
```

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();
for i = 1:length(sonnets)
```

```

sonnetWords = [sonnetWords ; split(sonnets(i))];
end
sonnetWords(1:10)

ans = 10x1 string
    "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);

```

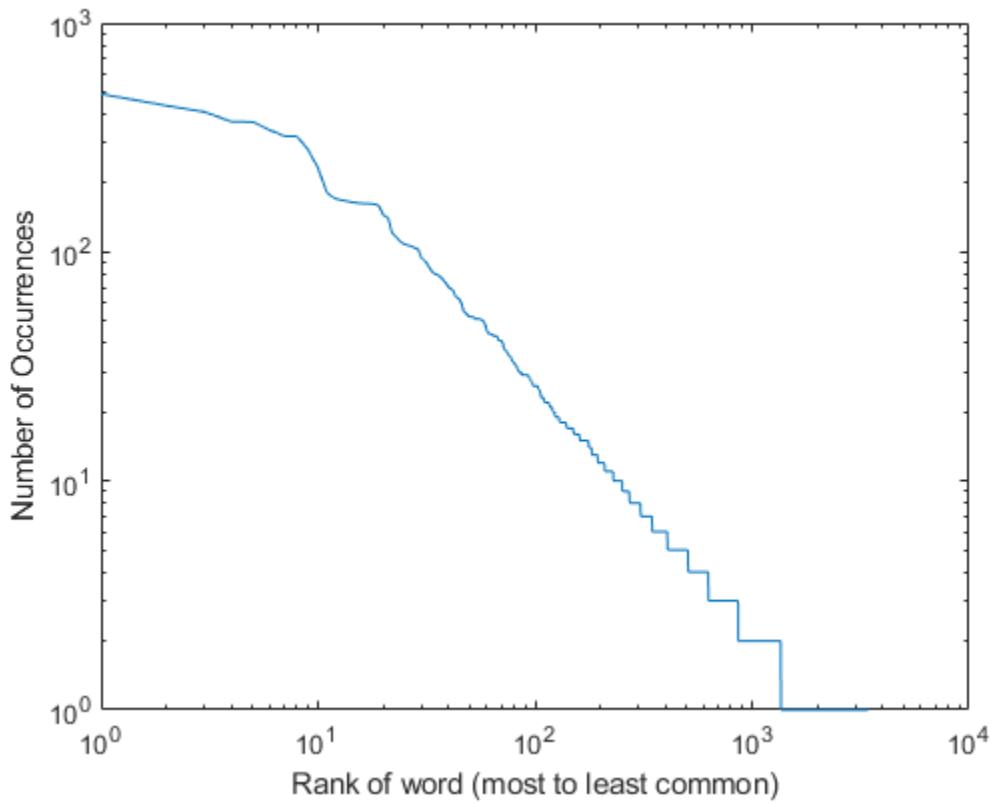
Plot Word Frequency

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
"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,:)
```

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 6-5
- “Search and Replace Text” on page 6-37
- “Compare Text” on page 6-32
- “Test for Empty Strings and Missing Values” on page 6-20

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  
""    ""    ""
```

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
    "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
      ""      "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

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

Related Examples

- “Create String Arrays” on page 6-5
- “Analyze Text Data with String Arrays” on page 6-15
- “Search and Replace Text” on page 6-37
- “Compare Text” on page 6-32

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.

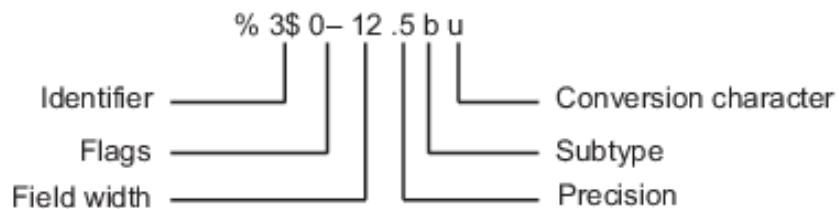
```
txt = sprintf('Displaying pi: \n %f \n %.2f \n %12f', A)

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.

Specifier	Description
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.	<code>%-5.2d</code>
Plus sign (+)	For numeric values, always print a leading sign character (+ or -). For text values, right-justify the converted argument in its field.	<code>%+5.2d</code> <code>%+5s</code>
Space	Insert a space before the value.	<code>% 5.2f</code>
Zero (0)	Pad with zeroes rather than spaces.	<code>%05.2f</code>
Pound sign (#)	Modify selected numeric conversions: <ul style="list-style-type: none">For <code>%o</code>, <code>%x</code>, or <code>%X</code>, print <code>0</code>, <code>0x</code>, or <code>0X</code> prefix.For <code>%f</code>, <code>%e</code>, or <code>%E</code>, print decimal point even when precision is 0.For <code>%g</code> or <code>%G</code>, do not remove trailing zeroes or decimal point.	<code>%#5.0f</code>

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 = '1st 2nd 3rd'</code>	<code>ans = '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	' '
Percent character	%%
Backslash	\\"

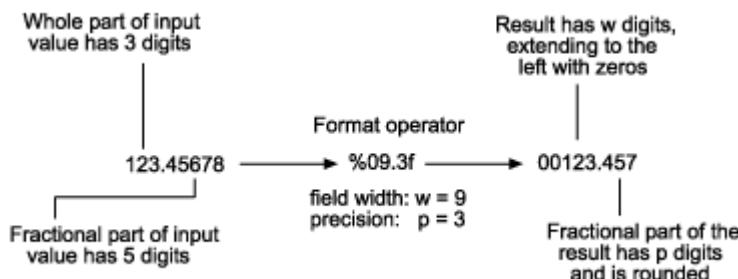
Special Character	Representation in Format Specifier
Alarm	\a
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v
Character whose Unicode numeric value can be represented by the hexadecimal number, N	\xN Example: sprintf('\x5A') returns 'Z'
Character whose Unicode numeric value can be represented by the octal number, N	\N Example: sprintf('\132') 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
%*f	Specify width as the following input argument, 15.
%.*f	Specify precision as the following input argument, 3.
%.*.*f	Specify width and precision as the following input arguments, 6, and 4.

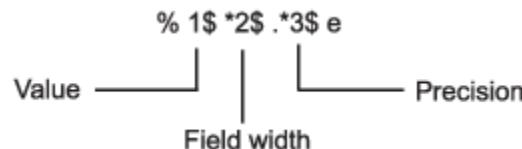
You can mix the two styles. For example, get the field width from the following input argument and the precision from the format specifier.

```
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
%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
v = [1.4 2.7 3.1]; sprintf('%4f %.4f %.4f',v) ans = '1.4000 2.7000 3.1000'	v = [1.4 2.7 3.1]; sprintf('%3\$.4f %1\$.4f %2\$.4f',v) ans = 1x0 empty char array

See Also

[compose](#) | [fprintf](#) | [num2str](#) | [sprintf](#)

Related Examples

- “Convert Text to Numeric Values” on page 6-49
- “Convert Numeric Values to Text” on page 6-45

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
"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
"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  
"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  
"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  
"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  
"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 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0

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

See Also

`eq | ge | gt | isletter | isspace | isstrprop | le | lt | ne | sort | strcmp`

Related Examples

- “Text in String and Character Arrays” on page 6-2
 - “Create String Arrays” on page 6-5
 - “Analyze Text Data with String Arrays” on page 6-15
 - “Search and Replace Text” on page 6-37
 - “Test for Empty Strings and Missing Values” on page 6-20

Search and Replace Text

Since R2016b. Replaces Searching and Replacing (R2016a).

Processing text data often involves finding and replacing substrings. There are several functions that find text and return different information: some functions confirm that the text exists, while others count occurrences, find starting indices, or extract substrings. These functions work on character vectors and string scalars, such as "yes", as well as character and string arrays, such as ["yes","no";"abc","xyz"]. In addition, you can use patterns to define rules for searching, such as one or more letter or digit characters.

Search for Text

To determine if text is present, use a function that returns logical values, like `contains`, `startsWith`, or `endsWith`. Logical values of 1 correspond to true, and 0 corresponds to false.

```
txt = "she sells seashells by the seashore";
TF = contains(txt, "sea")

TF = logical
1
```

Calculate how many times the text occurs using the `count` function.

```
n = count(txt, "sea")
n = 2
```

To locate where the text occurs, use the `strfind` function, which returns starting indices.

```
idx = strfind(txt, "sea")
idx = 1×2
11     28
```

Find and extract text using extraction functions, such as `extract`, `extractBetween`, `extractBefore`, or `extractAfter`.

```
mid = extractBetween(txt, "sea", "shore")
mid =
"shells by the sea"
```

Optionally, include the boundary text.

```
mid = extractBetween(txt, "sea", "shore", "Boundaries", "inclusive")
mid =
"seashells by the seashore"
```

Find Text in Arrays

The search and replacement functions can also find text in multi-element arrays. For example, look for color names in several song titles.

```
songs = ["Yellow Submarine";
    "Penny Lane";
    "Blackbird"];

colors =[ "Red", "Yellow", "Blue", "Black", "White"];

TF = contains(songs,colors)

TF = 3x1 logical array

1
0
1
```

To list the songs that contain color names, use the logical TF array as indices into the original songs array. This technique is called *logical indexing*.

```
colorful = songs(TF)

colorful = 2x1 string
    "Yellow Submarine"
    "Blackbird"
```

Use the function `replace` to replace text in `songs` that matches elements of `colors` with the string "Orange".

```
replace(songs,colors,"Orange")

ans = 3x1 string
    "Orange Submarine"
    "Penny Lane"
    "Orangebird"
```

Match Patterns

Since R2020b

In addition to searching for literal text, like "sea" or "yellow", you can search for text that matches a pattern. There are many predefined patterns, such as `digitsPattern` to find numeric digits.

```
address = "123a Sesame Street, New York, NY 10128";
nums = extract(address,digitsPattern)

nums = 2x1 string
    "123"
    "10128"
```

For additional precision in searches, you can combine patterns. For example, locate words that start with the character "S". Use a string to specify the "S" character, and `lettersPattern` to find additional letters after that character.

```
pat = "S" + lettersPattern;
StartWithS = extract(address,pat)

StartWithS = 2x1 string
    "Sesame"
```

```
"Street"
```

For more information, see “Build Pattern Expressions” on page 6-40.

See Also

`extract | count | pattern | replace | strfind | contains`

Related Examples

- “Text in String and Character Arrays” on page 6-2
- “Build Pattern Expressions” on page 6-40
- “Test for Empty Strings and Missing Values” on page 6-20
- “Regular Expressions” on page 2-51

Build Pattern Expressions

Since R2020b

Patterns are a tool to aid in searching for and modifying text. Similar to regular expressions, a pattern defines rules for matching text. Patterns can be used with text-searching functions like `contains`, `matches`, and `extract` to specify which portions of text these functions act on. You can build a pattern expression in a way similar to how you would build a mathematical expression, using pattern functions, operators, and literal text. Because building pattern expressions is open ended, patterns can become quite complicated. Building patterns in steps and using functions like `maskedPattern` and `namedPattern` can help organize complicated patterns.

Building Simple Patterns

The simplest pattern is built from a single pattern function. For example, `lettersPattern` matches any letter characters. There are many pattern functions for matching different types of characters and other features of text. A list of these functions can be found on the [pattern](#) reference page.

```
txt = "abc123def";
pat = lettersPattern;
extract(txt,pat)

ans = 2x1 string
    "abc"
    "def"
```

Patterns combine with other patterns and literal text by using the `plus (+)` operator. This operator appends patterns and text together in the order they are defined in the pattern expression. The combined patterns only match text in the same order. In this example, "YYYY/MM/DD" is not a match because a four-letter string must be at the end of the text.

```
txt = "Dates can be expressed as MM/DD/YYYY, DD/MM/YYYY, or YYYY/MM/DD";
pat = lettersPattern(2) + "/" + lettersPattern(2) + "/" + lettersPattern(4);
extract(txt,pat)

ans = 2x1 string
    "MM/DD/YYYY"
    "DD/MM/YYYY"
```

Patterns used with the `or (|)` operator specify that only one of the two specified patterns needs to match a section of text. If neither pattern is able to match then the pattern expression fails to match.

```
txt = "123abc";
pat = lettersPattern|digitsPattern;
extract(txt,pat)

ans = 2x1 string
    "123"
    "abc"
```

Some pattern functions take patterns as their input and modify them in some way. For example, `optionalPattern` makes a specified pattern match if possible, but the pattern is not required for a successful match.

```
txt = ["123abc" "abc"];
pat = optionalPattern(digitsPattern) + lettersPattern;
extract(txt,pat)

ans = 1x2 string
    "123abc"      "abc"
```

Boundary Patterns

Boundary patterns are a special type of pattern that do not match characters but rather match the boundaries between a designated character type and other characters or the start or end of that piece of text. For example, `digitBoundary` matches the boundaries between digit characters and nondigit characters and between digit characters and the start or end of the text. It does not match digit characters themselves. Boundary patterns are useful as delimiters for functions like `split`.

```
txt = "123abc";
pat = digitBoundary;
split(txt,pat)

ans = 3x1 string
    ""
    "123"
    "abc"
```

Boundary patterns are special amongst patterns because they can be negated using the `not(~)` operator. When negated in this way, boundary patterns match before or after characters that did not satisfy the requirements above. For example, `~digitBoundary` matches the boundary between:

- characters that are both digits
- characters that are both nondigits
- a nondigit character and the start or end of a piece of text

Use `replace` to mark the locations matched by `~digitBoundary` with a " | " character.

```
txt = "123abc";
pat = ~digitBoundary;
replace(txt,pat,"|")

ans =
"1|2|3a|b|c|"
```

Building Complicated Patterns in Steps

Sometimes a simple pattern is not sufficient to solve a problem and a more complicated pattern is needed. As a pattern expression grows it can become difficult to understand what it is matching. One way to simplify building a complicated pattern is building each part of the pattern separately and then combining the parts together into a single pattern expression.

For instance, email addresses use the form `local_part@domain.TLD`. Each of the three identifiers — `local_part`, `domain`, and `TLD` — must be a combination of digits, letters and underscore characters. To build the full pattern, start by defining a pattern for the identifiers. Build a pattern that matches one letter or digit character or one underscore character.

```
identCharacters = alphanumericsPattern(1) | "_";
```

Now, use `asManyOfPattern` to match one or more consecutive instances of `identCharacters`.

```
identifier = asManyOfPattern(identCharacters,1);
```

Next, build a pattern that matches an email containing multiple identifiers.

```
emailPattern = identifier + "@" + identifier + "." + identifier;
```

Test the pattern by seeing how well it matches the following example emails.

```
exampleEmails = ["janedoe@mathworks.com"
    "abe.lincoln@whitehouse.gov"
    "alberteinstein@physics.university.edu"];
matches(exampleEmails,emailPattern)

ans = 3x1 logical array
```

```
1
0
0
```

The pattern fails to match several of the example emails even though all the emails are valid. Both the local_part and domain can be made of a series of identifiers that are separated by periods. Use the `identifier` pattern to build a pattern that is capable of matching a series of identifiers. `asManyOfPattern` matches as many concurrent appearances of the specified pattern as possible, but if there are none the rest of the pattern is still able to match successfully.

```
identifierSeries = asManyOfPattern(identifier + ".") + identifier;
```

Use this pattern to build a new `emailPattern` that can match all of the example emails.

```
emailPattern = identifierSeries + "@" + identifierSeries + "." + identifier;
matches(exampleEmails,emailPattern)
```

```
ans = 3x1 logical array
```

```
1
1
1
```

Organizing Pattern Display

Complex patterns can sometimes be difficult to read and interpret, especially by those you share them with who are unfamiliar with the pattern's structure. For example, when displayed, `emailPattern` is long and difficult to read.

```
emailPattern
```

```
emailPattern = pattern
Matching:
```

```
asManyOfPattern(asManyOfPattern(alphanumericsPattern(1) | "_",1) + ".") + asManyOfPattern(alph
```

Part of the issue with the display is that there are many repetitions of the `identifier` pattern. If the exact details of this pattern are not important to users of the pattern, then the display of the

`identifier` pattern can be concealed using `maskedPattern`. This function creates a new pattern where the display of `identifier` is masked and the variable name, "identifier", is displayed instead. Alternatively, you can specify a different name to be displayed. The details of patterns that are masked in this way can be accessed by clicking "Show all details" in the displayed pattern.

```
identifier = maskedPattern(identifier);
identifierSeries = asManyOfPattern(identifier + ".") + identifier

identifierSeries = pattern
Matching:
  asManyOfPattern(identifier + ".") + identifier

Show all details
```

Patterns can be further organized using the `namedPattern` function. `namedPattern` designates a pattern as a named pattern that changes how the pattern is displayed when combined with other patterns. Email addresses have several important portions, `local_part@domain.TLD`, which each have their own matching rules. Create a named pattern for each section.

```
localPart = namedPattern(identifierSeries, "local_part");
```

Named patterns can be nested, to further delineate parts of a pattern. To nest a named pattern, build a pattern using named patterns and then designate that pattern as a named pattern. For example, `Domain.TLD` can be divided into the domain, subdomains, and the top level domain (TLD). Create named patterns for each part of `domain.TLD`.

```
subdomain = namedPattern(identifierSeries, "subdomain");
domainName = namedPattern(identifier, "domainName");
tld = namedPattern(identifier, "TLD");
```

Nest the named patterns for the components of `domain` underneath a single named pattern `domain`.

```
domain = optionalPattern(subdomain + ".") + ...
         domainName + "."
         tld;
domain = namedPattern(domain);
```

Combine the patterns together into a single named pattern, `emailPattern`. In the display of `emailPattern` you can see each named pattern and what they match as well as the information on any nested named patterns.

```
emailPattern = localPart + "@" + domain

emailPattern = pattern
Matching:
  local_part + "@" + domain
```

Using named patterns:

```
local_part : asManyOfPattern(identifier + ".") + identifier
domain     : optionalPattern(subdomain + ".") + domainName + "."
            + TLD
subdomain : asManyOfPattern(identifier + ".") + identifier
domainName: identifier
TLD       : identifier
```

Show all details

You can access named patterns and nested named patterns by dot-indexing into a pattern. For example, you can access the nested named pattern `subdomain` by dot-indexing from `emailPattern` into `domain` and then dot-indexing again into `subdomain`.

`emailPattern.domain.subdomain`

`ans = pattern`
Matching:

`asManyOfPattern(identifier + ".") + identifier`

Show all details

Dot-assignment can be used to change named patterns without needing to rewrite the rest of the pattern expression.

`emailPattern.domain = "mathworks.com"`

`emailPattern = pattern`
Matching:

`local_part + "@" + domain`

Using named patterns:

`local_part: asManyOfPattern(identifier + ".") + identifier`
`domain : "mathworks.com"`

Show all details

See Also

`pattern` | `string` | `regexp` | `contains` | `replace` | `extract`

More About

- “Search and Replace Text” on page 6-37
- “Regular Expressions” on page 2-51

Convert Numeric Values to Text

This example shows how to convert numeric values to text and append them to larger pieces of text. For example, you might want to add a label or title to a plot, where the label includes a number that describes a characteristic of the plot.

Convert to Strings

Before R2016b, convert to character vectors using num2str.

To convert a number to a string that represents it, use the `string` function.

```
str = string(pi)

str =
"3.1416"
```

The `string` function converts a numeric array to a string array having the same size.

```
A = [256 pi 8.9e-3];
str = string(A)

str = 1x3 string
"256"      "3.141593"      "0.0089"
```

You can specify the format of the output text using the `compose` function, which accepts format specifiers for precision, field width, and exponential notation.

```
str = compose("%9.7f",pi)

str =
"3.1415927"
```

If the input is a numeric array, then `compose` returns a string array. Return a string array that represents numbers using exponential notation.

```
A = [256 pi 8.9e-3];
str = compose("%5.2e",A)

str = 1x3 string
"2.56e+02"      "3.14e+00"      "8.90e-03"
```

Add Numbers to Strings

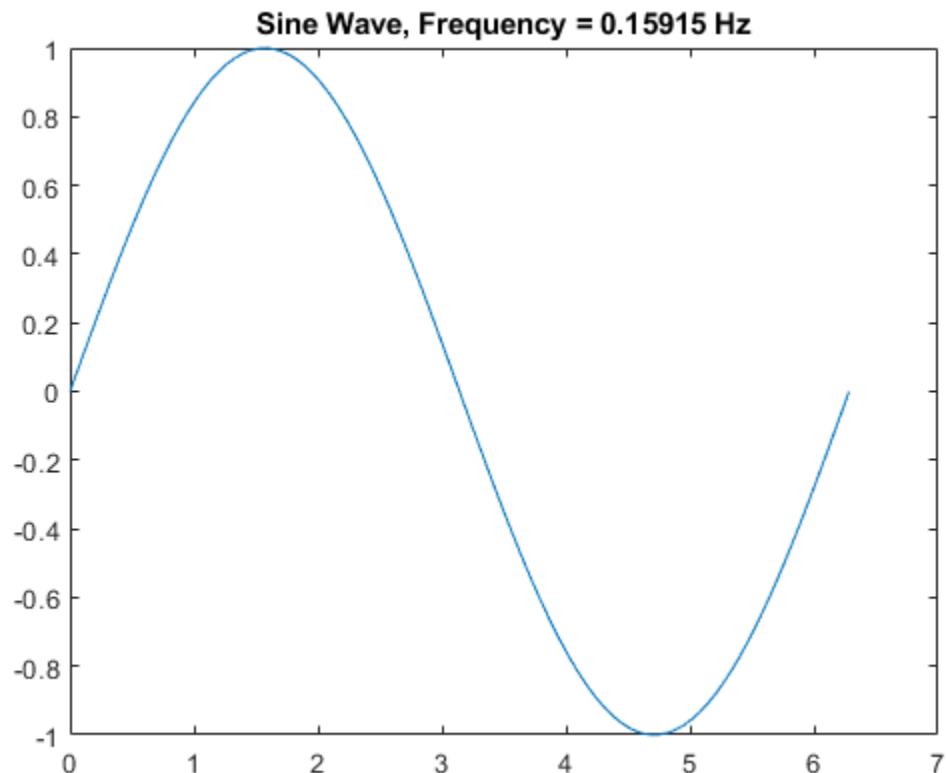
Before R2016b, convert numbers to character vectors and concatenate characters in brackets, [].

The simplest way to combine text and numbers is to use the `plus` operator (+). This operator automatically converts numeric values to strings when the other operands are strings.

For example, plot a sine wave. Calculate the frequency of the wave and add a string representing that value in the title of the plot.

```
X = linspace(0,2*pi);
Y = sin(X);
plot(X,Y)
```

```
freq = 1/(2*pi);
str = "Sine Wave, Frequency = " + freq + " Hz"
str =
"Sine Wave, Frequency = 0.15915 Hz"
title(str)
```



Sometimes existing text is stored in character vectors or cell arrays of character vectors. However, the `plus` operator also automatically converts those types of data to strings when another operand is a string. To combine numeric values with those types of data, first convert the numeric values to strings, and then use `plus` to combine the text.

```
str = 'Sine Wave, Frequency = ' + string(freq) + {' Hz'}
str =
"Sine Wave, Frequency = 0.15915 Hz"
```

Character Codes

If your data contains integers that represent Unicode® values, use the `char` function to convert the values to the corresponding characters. The output is a character vector or array.

```
u = [77 65 84 76 65 66];
c = char(u)

c =
'MATLAB'
```

Converting Unicode values also allows you to include special characters in text. For instance, the Unicode value for the degree symbol is 176. To add `char(176)` to a string, use `plus`.

```
deg = char(176);
temp = 21;
str = "Temperature: " + temp + deg + "C"

str =
"Temperature: 21°C"
```

Before R2016b, use `num2str` to convert the numeric value to a character vector, and then concatenate.

```
str = ['Temperature: ' num2str(temp) deg 'C']

str =
'Temperature: 21°C'
```

Hexadecimal and Binary Values

Since R2019b

You can represent hexadecimal and binary values in your code either using text or using *literals*. The recommended way to represent them is to write them as literals. You can write hexadecimal and binary literals using the `0x` and `0b` prefixes respectively. However, it can sometimes be useful to represent such values as text, using the `dec2hex` or `dec2bin` functions.

For example, set a bit in a binary value. If you specify the binary value using a literal, then it is stored as an integer. After setting one of the bits, display the new binary value as text using the `dec2bin` function.

```
register = 0b10010110

register = uint8
    150

register = bitset(register,5,0)

register = uint8
    134

binStr = dec2bin(register)

binStr =
'10000110'
```

See Also

`char` | `compose` | `dec2bin` | `dec2hex` | `plus` | `string`

More About

- “Convert Text to Numeric Values” on page 6-49
- “Hexadecimal and Binary Values” on page 6-55
- “Convert BetweenDatetime Arrays, Numbers, and Text” on page 7-42
- “Formatting Text” on page 6-24

- “Unicode and ASCII Values” on page 6-53

Convert Text to Numeric Values

This example shows how to convert text to the numeric values that it represents. Typically, you need to perform such conversions when you have text that represents numbers to be plotted or used in calculations. For example, the text might come from a text file or spreadsheet. If you did not already convert it to numeric values when importing it into MATLAB®, you can use the functions shown in this example.

You can convert string arrays, character vectors, and cell arrays of character vectors to numeric values. Text can represent hexadecimal or binary values, though when you convert them to numbers they are stored as decimal values. You can also convert text representing dates and time to `datetime` or `duration` values, which can be treated like numeric values.

Double-Precision Values

The recommended way to convert text to double-precision values is to use the `str2double` function. It can convert character vectors, string arrays, and cell arrays of character vectors.

For example, create a character vector using single quotes and convert it to the number it represents.

```
X = str2double('3.1416')
X = 3.1416
```

If the input argument is a string array or cell array of character vectors, then `str2double` converts it to a numeric array having the same size. You can create strings using double quotes. (Strings have the `string` data type, while character vectors have the `char` data type.)

```
str = ["2.718", "3.1416";
       "137", "0.015"]

str = 2x2 string
      "2.718"    "3.1416"
      "137"      "0.015"

X = str2double(str)

X = 2×2
      2.7180    3.1416
     137.0000    0.0150
```

The `str2double` function can convert text that includes commas (as thousands separators) and decimal points. For example, you can use `str2double` to convert the `Balance` variable in the table below. `Balance` represents numbers as strings, using a comma as the thousands separator.

```
load balances
balances

balances=3×2 table
  Customer        Balance
  _____
  "Diaz"         "13,790.00"
  "Johnson"      "2,456.10"
```

```
"Wu"      "923.71"
```

```
T.Balance = str2double(T.Balance)
```

```
T=3×2 table
  Customer      Balance
  _____
  "Diaz"        13790
  "Johnson"     2456.1
  "Wu"          923.71
```

If `str2double` cannot convert text to a number, then it returns a `NaN` value.

While the `str2num` function can also convert text to numbers, it is **not** recommended. `str2num` uses the `eval` function, which can cause unintended side effects when the text input includes a function name. To avoid these issues, use `str2double`.

As an alternative, you can convert strings to double-precision values using the `double` function. If the input is a string array, then `double` returns a numeric array that has the same size, just as `str2double` does. However, if the input is a character vector, then `double` converts the individual characters to numbers representing their Unicode values.

```
X = double("3.1416")
X = 3.1416
X = double('3.1416')
X = 1×6
      51    46    49    52    49    54
```

This list summarizes the best practices for converting text to numeric values.

- To convert text to numeric values, use the `str2double` function. It treats string arrays, character vectors, and cell arrays of character vectors consistently.
- You can also use the `double` function for string arrays. However, it treats character vectors differently.
- **Avoid** `str2num`. It calls the `eval` function which can have unintended consequences.

Hexadecimal and Binary Values

You can represent hexadecimal and binary numbers as text or as *literals*. When you write them as literals, you must use the `0x` and `0b` prefixes. When you represent them as text and then convert them, you can use the prefixes, but they are not required.

For example, write a hexadecimal number as a literal. The prefix is required.

```
D = 0x3FF
D = uint16
      1023
```

Then convert text representing the same value by using the `hex2dec` function. It recognizes the prefix but does not require it.

```
D = hex2dec('3FF')
D = 1023
D = hex2dec('0x3FF')
D = 1023
```

Convert text representing binary values using the `bin2dec` function.

```
D = bin2dec('101010')
D = 42
D = bin2dec('0b101010')
D = 42
```

Dates and Times

MATLAB provides the `datetime` and `duration` data types to store dates and times, and to treat them as numeric values. To convert text representing dates and times, use the `datetime` and `duration` functions.

Convert text representing a date to a `datetime` value. The `datetime` function recognizes many common formats for dates and times.

```
C = '2019-09-20'
C =
'2019-09-20'
D = datetime(C)
D = datetime
20-Sep-2019
```

You can convert arrays representing dates and times.

```
str = ["2019-01-31", "2019-02-28", "2019-03-31"]
str = 1x3 string
    "2019-01-31"      "2019-02-28"      "2019-03-31"

D = datetime(str)
D = 1x3 datetime
    31-Jan-2019      28-Feb-2019      31-Mar-2019
```

If you convert text to `duration` values, then use the `hh:mm:ss` or `dd:hh:mm:ss` formats.

```
D = duration('12:34:56')
```

```
D = duration  
12:34:56
```

See Also

`bin2dec | datetime | double | duration | hex2dec | str2double | table`

More About

- “Convert Numeric Values to Text” on page 6-45
- “Convert BetweenDatetime Arrays, Numbers, and Text” on page 7-42
- “Hexadecimal and Binary Values” on page 6-55
- “Formatting Text” on page 6-24
- “Unicode and ASCII Values” on page 6-53

Unicode and ASCII Values

MATLAB® stores all characters as Unicode® characters using the UTF-16 encoding, where every character is represented by a numeric code value. (Unicode incorporates the ASCII character set as the first 128 symbols, so ASCII characters have the same numeric codes in Unicode and ASCII.) Both character arrays and string arrays use this encoding. You can convert characters to their numeric code values by using various numeric conversion functions. You can convert numbers to characters using the `char` function.

Convert Characters to Numeric Code Values

You can convert characters to integers that represent their Unicode code values. To convert a single character or a character array, use any of these functions:

- `double`
- `uint16`, `uint32`, or `uint64`

The best practice is to use the `double` function. However, if you need to store the numeric values as integers, use unsigned integers having at least 16 bits because MATLAB uses the UTF-16 encoding.

Convert a character vector to Unicode code values using the `double` function.

```
C = 'MATLAB'
C =
'MATLAB'
unicodeValues = double(C)
unicodeValues = 1×6
    77     65     84     76     65     66
```

You cannot convert characters in a string array directly to Unicode code values. In particular, the `double` function converts strings to the numbers they represent, just as the `str2double` function does. If `double` cannot convert a string to a number, then it returns a `NaN` value.

```
str = "MATLAB";
double(str)
ans = NaN
```

To convert characters in a string, first convert the string to a character vector, or use curly braces to extract the characters. Then convert the characters using a function such as `double`.

```
C = char(str);
unicodeValues = double(C)
unicodeValues = 1×6
    77     65     84     76     65     66
```

Convert Numeric Code Values to Characters

You can convert Unicode values to characters using the `char` function.

```
D = [77 65 84 76 65 66]
D = 1×6
    77     65     84     76     65     66

C = char(D)
C =
'MATLAB'
```

A typical use for `char` is to create characters you cannot type and append them to strings. For example, create the character for the degree symbol and append it to a string. The Unicode code value for the degree symbol is 176.

```
deg = char(176)
deg =
    °
myLabel = append("Current temperature is 21",deg,"C")
myLabel =
"Current temperature is 21°C"
```

For more information on Unicode, including mappings between characters and code values, see [Unicode](#).

See Also

`char` | `double` | `int16` | `int32` | `int64` | `int8` | `single` | `string` | `uint16` | `uint32` | `uint64` | `uint8`

More About

- “Convert Text to Numeric Values” on page 6-49
- “Convert Numeric Values to Text” on page 6-45

External Websites

- [Unicode](#)

Hexadecimal and Binary Values

You can represent numbers as hexadecimal or binary values. In some contexts, these representations of numbers are more convenient. For example, you can represent the bits of a hardware register using binary values. In MATLAB®, there are two ways to represent hexadecimal and binary values:

- As *literals*. Starting in R2019b, you can write hexadecimal and binary values as literals using an appropriate prefix as notation. For example, `0x2A` is a literal that specifies 42—and MATLAB stores it as a number, *not* as text.
- As strings or character vectors. For example, the character vector `'2A'` represents the number 42 as a hexadecimal value. When you represent a hexadecimal or binary value using text, enclose it in quotation marks. MATLAB stores this representation as text, not a number.

MATLAB provides several functions for converting numbers to and from their hexadecimal and binary representations.

Write Integers Using Hexadecimal and Binary Notation

Hexadecimal literals start with a `0x` or `0X` prefix, while binary literals start with a `0b` or `0B` prefix. MATLAB stores the number written with this notation as an integer. For example, these two literals both represent the integer 42.

```
A = 0x2A
A = uint8
    42

B = 0b101010
B = uint8
    42
```

Do not use quotation marks when you write a number using this notation. Use `0-9`, `A-F`, and `a-f` to represent hexadecimal digits. Use `0` and `1` to represent binary digits.

By default, MATLAB stores the number as the smallest unsigned integer type that can accommodate it. However, you can use an optional suffix to specify the type of integer that stores the value.

- To specify unsigned 8-, 16-, 32-, and 64-bit integer types, use the suffixes `u8`, `u16`, `u32`, and `u64`.
- To specify signed 8-, 16-, 32-, and 64-bit integer types, use the suffixes `s8`, `s16`, `s32`, and `s64`.

For example, write a hexadecimal literal to be stored as a signed 32-bit integer.

```
A = 0x2As32
A = int32
    42
```

When you specify signed integer types, you can write literals that represent negative numbers. Represent negative numbers in two's complement form. For example, specify a negative number with a literal using the `s8` suffix.

```
A = 0xFFs8
A = int8
    -1
```

Since MATLAB stores these literals as numbers, you can use them in any context or function where you use numeric arrays.

Represent Hexadecimal and Binary Values as Text

You can also convert integers to character vectors that represent them as hexadecimal or binary values using the `dec2hex` and `dec2bin` functions. Convert an integer to hexadecimal.

```
hexStr = dec2hex(255)  
hexStr =  
'FF'
```

Convert an integer to binary.

```
binStr = dec2bin(16)  
binStr =  
'10000'
```

Since these functions produce text, use them when you need text that represents numeric values. For example, you can append these values to a title or a plot label, or write them to a file that stores numbers as their hexadecimal or binary representations.

Represent Arrays of Hexadecimal Values as Text

The recommended way to convert an array of numbers to text is to use the `compose` function. This function returns a string array having the same size as the input numeric array. To produce hexadecimal format, use `%X` as the format specifier.

```
A = [255 16 12 1024 137]  
A = 1×5  
255 16 12 1024 137  
  
hexStr = compose("%X",A)  
hexStr = 1x5 string  
"FF" "10" "C" "400" "89"
```

The `dec2hex` and `dec2bin` functions also convert arrays of numbers to text representing them as hexadecimal or binary values. However, these functions return character arrays, where each row represents a number from the input numeric array, padded with zeros as necessary.

Convert Binary Representations to Hexadecimal

To convert a binary value to hexadecimal, start with a binary literal, and convert it to text representing its hexadecimal value. Since a literal is interpreted as a number, you can specify it directly as the input argument to `dec2hex`.

```
D = 0b1111;  
hexStr = dec2hex(D)  
  
hexStr =  
'F'
```

If you start with a hexadecimal literal, then you can convert it to text representing its binary value using `dec2bin`.

```
D = 0x8F;
binStr = dec2bin(D)

binStr =
'10001111'
```

Bitwise Operations with Binary Values

One typical use of binary numbers is to represent bits. For example, many devices have registers that provide access to a collection of bits representing data in memory or the status of the device. When working with such hardware you can use numbers in MATLAB to represent the value in a register. Use binary values and bitwise operations to represent and access particular bits.

Create a number that represents an 8-bit register. It is convenient to start with binary representation, but the number is stored as an integer.

```
register = 0b10010110

register = uint8
150
```

To get or set the values of particular bits, use bitwise operations. For example, use the `bitand` and `bitshift` functions to get the value of the fifth bit. (Shift that bit to the first position so that MATLAB returns a 0 or 1. In this example, the fifth bit is a 1.)

```
b5 = bitand(register,0b10000);
b5 = bitshift(b5,-4)

b5 = uint8
1
```

To flip the fifth bit to 0, use the `bitset` function.

```
register = bitset(register,5,0)

register = uint8
134
```

Since `register` is an integer, use the `dec2bin` function to display all the bits in binary format. `binStr` is a character vector, and represents the binary value without a leading `0b` prefix.

```
binStr = dec2bin(register)

binStr =
'10000110'
```

See Also

`bin2dec` | `bitand` | `bitset` | `bitshift` | `dec2bin` | `dec2hex` | `hex2dec` | `sprintf` | `sscanf`

More About

- “Convert Text to Numeric Values” on page 6-49
- “Convert Numeric Values to Text” on page 6-45

- “Formatting Text” on page 6-24
- “Bit-Wise Operations” on page 2-38
- “Perform Cyclic Redundancy Check” on page 2-44

External Websites

- Two's Complement

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 6-5.

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 = strlength(str)

L = 11
```

You also can use `strlength` 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 = strlength("")

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 an 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 6-5
- “Test for Empty Strings and Missing Values” on page 6-20
- “Compare Text” on page 6-32
- “Update Your Code to Accept Strings” on page 6-64

Update Your Code to Accept Strings

In R2016b, MATLAB introduced string arrays as a data type for text. As of R2018b, all MathWorks products are *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 6-70.

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 strings 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 homogeneous 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{1} = convertCharsToStrings(varargin{1});
    ...

```

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>

Required Input Argument Type	Old Check	New Check
		<code>validateattributes(X, {'char', 'string'}, {'scalarmtext'})</code>
Character vector or string scalar	<code>validateattributes(X, {'char'}, {'row'})</code>	<code>validateattributes(X, {'char', 'string'}, {'scalarmtext'})</code>
Nonempty character vector or string scalar	<code>ischar(X) && ~isempty(X)</code>	<code>(ischar(X) isStringScalar(X)) && strlength(X) ~= 0</code> <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 | issstring | string | strings | strlength | validateattributes`

More About

- “Create String Arrays” on page 6-5
- “Test for Empty Strings and Missing Values” on page 6-20
- “Compare Text” on page 6-32
- “Search and Replace Text” on page 6-37
- “Frequently Asked Questions About String Arrays” on page 6-59

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
eval	Execute a MATLAB expression.
sscanf	Read a character array under format control.

Functions to Search or Compare Character Arrays

Function	Description
<code>regexp</code>	Match regular expression on page 2-51.
<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 7-5
- “Convert Date and Time to Julian Date or POSIX Time” on page 7-7
- “Set Date and Time Display Format” on page 7-10
- “Generate Sequence of Dates and Time” on page 7-14
- “Share Code and Data Across Locales” on page 7-20
- “Extract or Assign Date and Time Components of Datetime Array” on page 7-23
- “Combine Date and Time from Separate Variables” on page 7-26
- “Date and Time Arithmetic” on page 7-28
- “Compare Dates and Time” on page 7-33
- “Plot Dates and Durations” on page 7-36
- “Core Functions Supporting Date and Time Arrays” on page 7-41
- “Convert Between Datetime Arrays, Numbers, and Text” on page 7-42
- “Carryover in Date Vectors and Strings” on page 7-47
- “Converting Date Vector Returns Unexpected Output” on page 7-48

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.

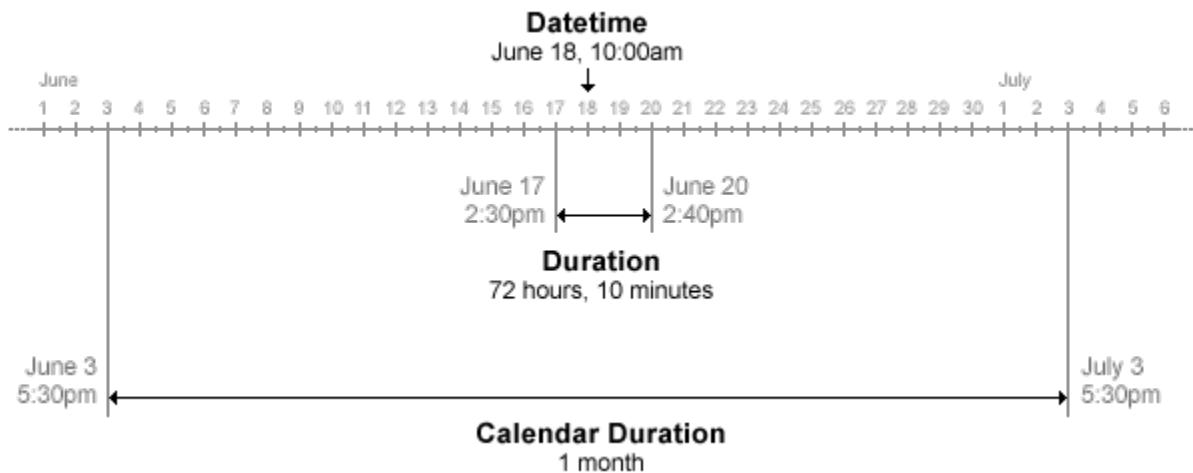
```
whos t2
```

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 7-7

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
24-Feb-2021 01:14:15
```

`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
24-Feb-2021 01:14:15
```

As an alternative, you can specify the `TimeZone` property after you create the array.

```
d.TimeZone = 'America/Los_Angeles'
d = datetime
23-Feb-2021 22:14:15
```

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
    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    0.1666666666511446    0.1666666666511446
```

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
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 = 1x3
```

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 = 1x3
```

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 7-5

Set Date and Time Display Format

In this section...

- “Formats for Individual Date and Duration Arrays” on page 7-10
- “datetime Display Format” on page 7-10
- “duration Display Format” on page 7-11
- “calendarDuration Display Format” on page 7-11
- “Default datetime Format” on page 7-12

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.
'defaultdate'	Use the default date display format that does not show time components.

To change the default formats, see “Default datetime Format” on page 7-12.

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

Value of Format	Example
'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
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 '`defauldate`'. 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('defauldate', '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 7-14
- “Add Duration or Calendar Duration to Create Sequence of Dates” on page 7-16
- “Specify Length and Endpoints of Date or Duration Sequence” on page 7-17
- “Sequence of Datetime Values Using Calendar Rules” on page 7-17

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
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
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
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
    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
    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
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
    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
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
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
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
1mo    1mo    1mo    1mo
```

Determine the number of days between each date.

```
dt = caldiff(t, 'days')
```

```
dt = 1x4 calendarDuration
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
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
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
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  
24-Feb-2021 Wed
```

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  
01-Mar-2021 Mon 08-Mar-2021 Mon 15-Mar-2021 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  
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  
30-Apr-2014 31-May-2014 30-Jun-2014
```

Determine the number of days between each date.

```
dt = caldiff(t,'days')  
  
dt = 1x2 calendarDuration  
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  
24-Feb-2021 00:16:43  
  
t = dateshift(t1,'start','hour',0:4)  
t = 1x5 datetime  
Columns 1 through 3  
  
24-Feb-2021 00:00:00    24-Feb-2021 01:00:00    24-Feb-2021 02:00:00  
  
Columns 4 through 5  
  
24-Feb-2021 03:00:00    24-Feb-2021 04:00:00
```

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  
23-Feb-2021 23:00:00    24-Feb-2021 00:00:00    24-Feb-2021 01:00:00
```

See Also

`dateshift` | `linspace`

Share Code and Data Across Locales

In this section...

- “Write Locale-Independent Date and Time Code” on page 7-20
- “Write Dates in Other Languages” on page 7-21
- “Read Dates in Other Languages” on page 7-21

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  
2021-02-24
```

instead of this:

```
t = datetime('today','Format','eeee, dd-MMM-yyyy')  
t = datetime  
Wednesday, 24-Feb-2021
```

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  
00:49
```

instead of this:

```
t = datetime('now','Format','hh:mm a')  
t = datetime  
12:49 AM
```

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  
24-02-2021 -0500
```

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  
24-Feb-2021  
25-Feb-2021
```

```
S = cellstr(t,'dd. MMMM yyyy','de_DE')
```

```
S = 2x1 cell  
{'24. Februar 2021'}  
{'25. Februar 2021'}
```

`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
    24-Feb-2021 20:22:22    25-Mar-2022 16:22:22    26-Apr-2023 12:22:22
```

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
    2021      2022      2023
```

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
    2      3      4
```

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
    2      3      4
```

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
  {'February'}    {'March'}    {'April'}
```

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
  9      13      17
```

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
  2021      2022      2023

m = 1x3
  2      3      4

d = 1x3
  24      25      26
```

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 = 1x3
  20      16      12

m = 1x3
  22      22      22

s = 1x3
  22.8550  22.8550  22.8550
```

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 = 1x3 datetime  
24-Feb-2014 20:22:22 25-Mar-2014 16:22:22 26-Apr-2014 12:22:22
```

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  
24-Jan-2014 20:22:22 25-Feb-2014 16:22:22 26-Mar-2014 12:22:22
```

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  
24-Jan-2014 25-Feb-2014 26-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  
30-Dec-2013 01-Feb-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  
24-Feb-2021 00:18:21
```

Find future points in time by adding a sequence of hours.

```
t2 = t1 + hours(1:3)  
t2 = 1x3 datetime  
24-Feb-2021 01:18:21 24-Feb-2021 02:18:21 24-Feb-2021 03:18:21
```

Verify that the difference between each pair of datetime values in `t2` is 1 hour.

```
dt = diff(t2)  
dt = 1x2 duration  
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  
23-Feb-2021 23:58:21 23-Feb-2021 23:48:21 23-Feb-2021 23:38:21
```

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  
25-Feb-2021 00:18:21 26-Feb-2021 00:18:21 27-Feb-2021 00:18:21
```

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
```

Find future points in time by adding a sequence of fixed-length (24-hour) days.

```
t2 = t1 + days(0:2)

t2 = 1x3 datetime
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
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
08-Mar-2014 09-Mar-2014 10-Mar-2014
```

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
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  
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  
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  
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  
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
25-Feb-2021 00:18:22 26-Feb-2021 00:18:22 27-Feb-2021 00:18:22
```

```
t1 = datetime('yesterday')
```

```
t1 = datetime
23-Feb-2021
```

```
dt = t2 - t1
```

```
dt = 1x3 duration
    48:18:22    72:18:22    96:18:22
```

```
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
    2.0128 days   3.0128 days   4.0128 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
    2.4153 days   3.6153 days   4.8153 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
24-Feb-2021
```

```
t2 = t1 + calmonths(0:2) + caldays(4)
```

```
t2 = 1x3 datetime
28-Feb-2021   28-Mar-2021   28-Apr-2021
```

```
dt = between(t1,t2)
```

```
dt = 1x3 calendarDuration
    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
    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 >= '26-Sep-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  
02:30:30  
03:15:00
```

```
B = duration([2,40,0;2,50,0])
```

```
B = 2x1 duration  
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
    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 2-29

Plot Dates and Durations

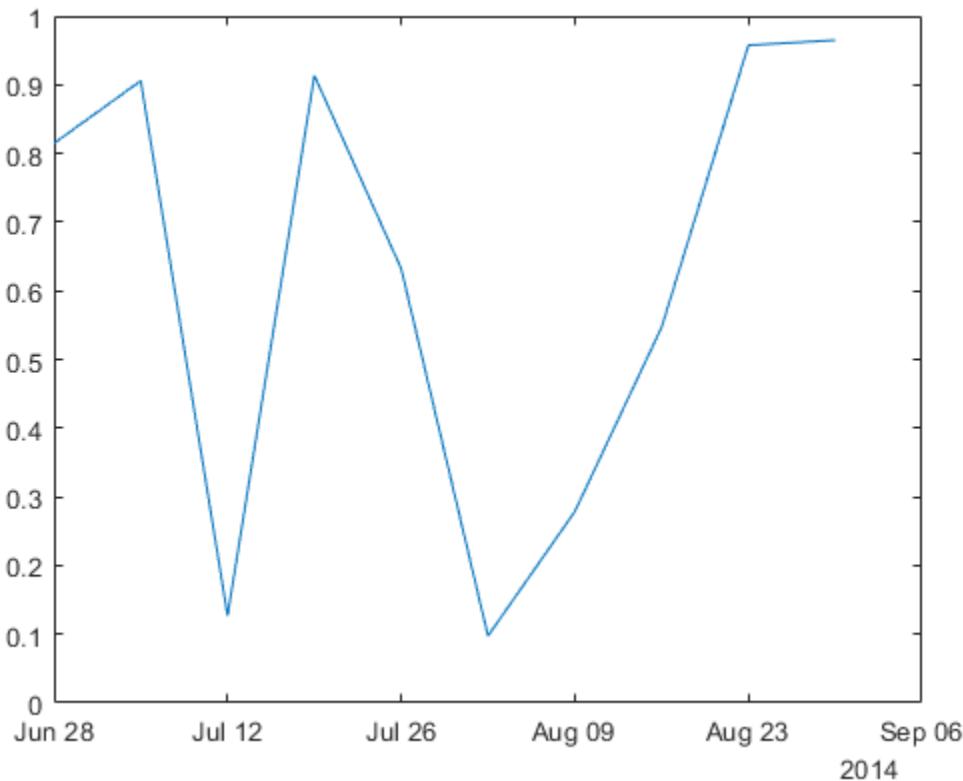
You can create plots of datetime and duration values with a variety of graphics functions. You also can customize the axes, such as changing the format of the tick labels or changing the axis limits.

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.

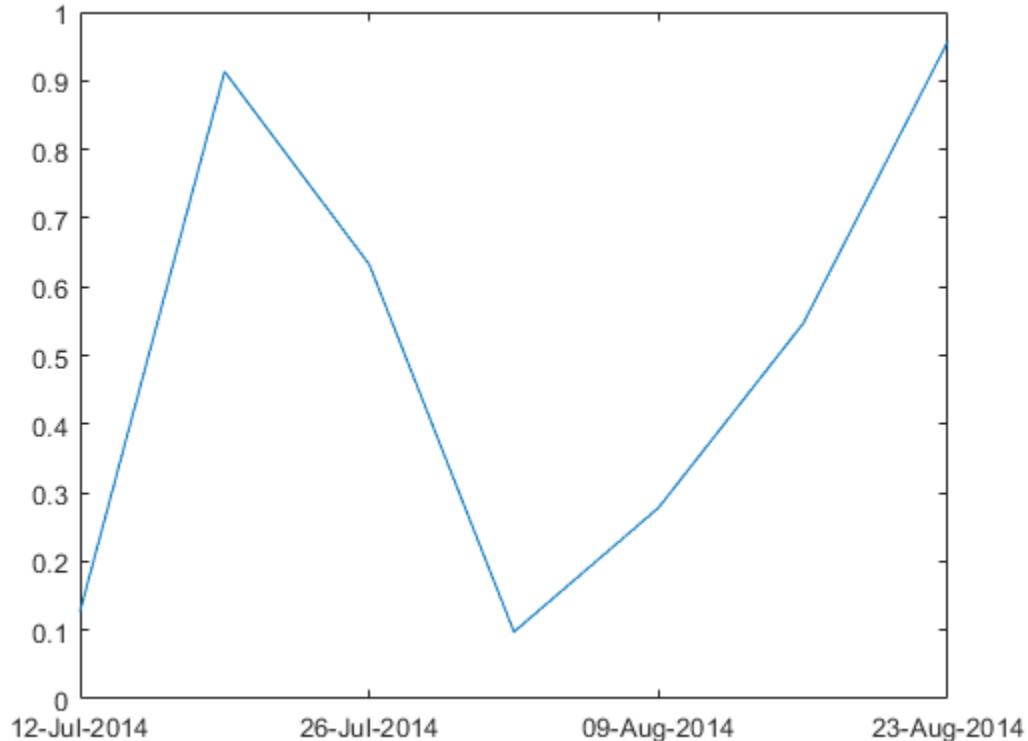
```
t = datetime(2014,6,28) + calweeks(0:9);
y = rand(1,10);
plot(t,y);
```



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')
```

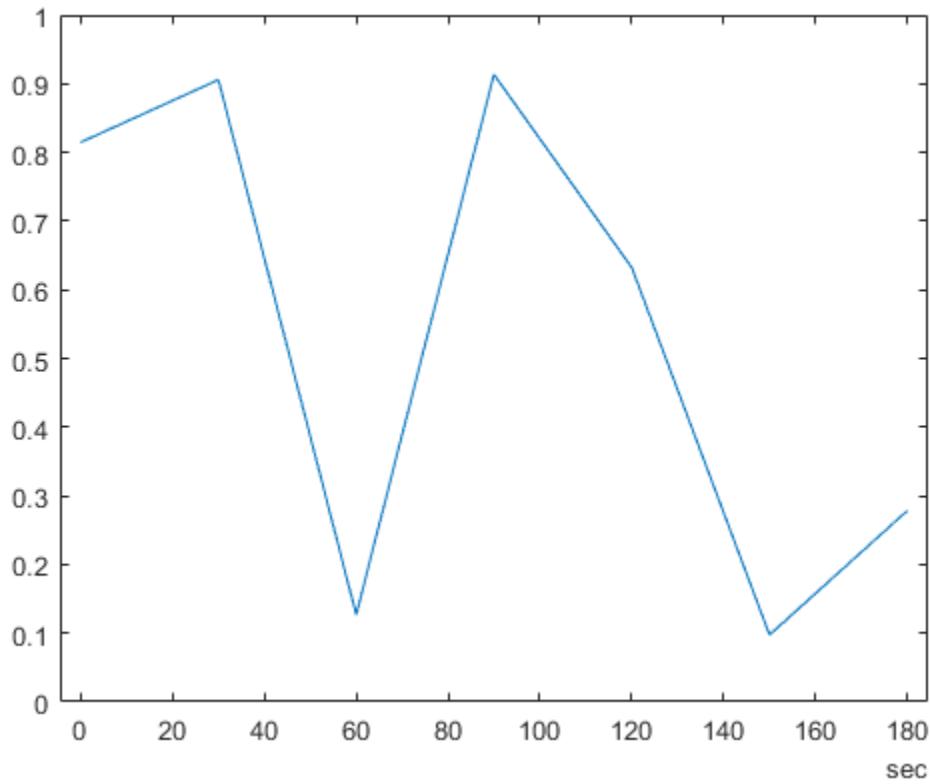


Line Plot with Durations

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);
```

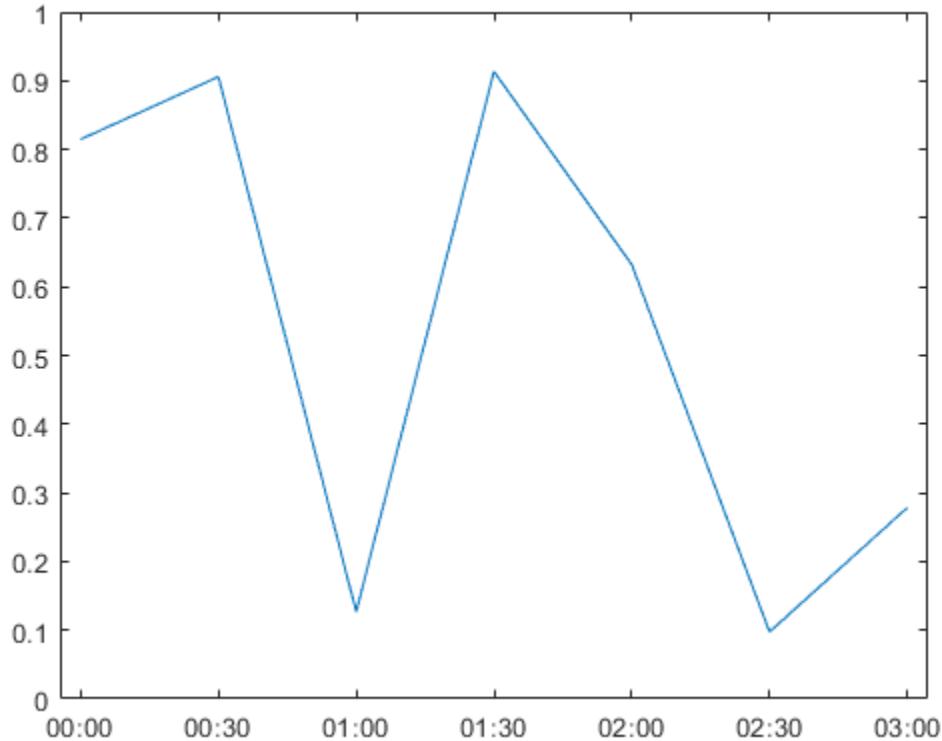


View the x-axis limits. Since the duration tick labels are in terms of a single unit (minutes), the limits are stored in terms of that unit.

```
xl = xlim  
xl = 1x2 duration  
-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')
```



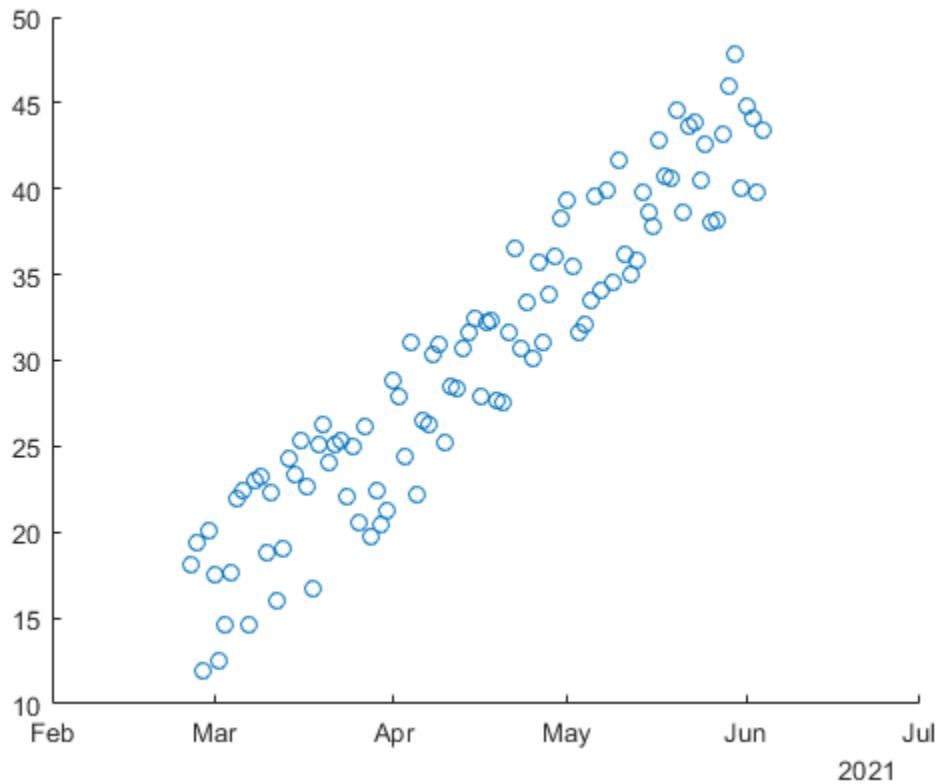
View the x-axis limits again. Since the duration tick labels are now in terms of multiple units, the limits are stored in units of 24-hour days.

```
xl = xlim  
xl = 1x2 duration  
-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
semilogx (x values must be numeric)	semilogy (y values must be numeric)
stem	stairs
scatter	scatter3
area	mesh
surf	surface
fill	fill3
line	text
histogram	

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.

<code>size</code>	<code>isequal</code>	<code>intersect</code>	<code>plus</code>	<code>plot</code>
<code>length</code>	<code>isequaln</code>	<code>ismember</code>	<code>minus</code>	<code>plot3</code>
<code>ndims</code>		<code>setdiff</code>	<code>uminus</code>	<code>scatter</code>
<code>numel</code>	<code>eq</code>	<code>setxor</code>	<code>times</code>	<code>scatter3</code>
<code>isrow</code>	<code>ne</code>	<code>unique</code>	<code>rdivide</code>	<code>bar</code>
<code>iscolumn</code>	<code>lt</code>	<code>union</code>	<code>ldivide</code>	<code>barh</code>
<code>cat</code>	<code>le</code>	<code>abs</code>	<code>mtimes</code>	<code>histogram</code>
<code>horzcat</code>	<code>ge</code>	<code>floor</code>	<code>mrdivide</code>	<code>stem</code>
<code>vertcat</code>	<code>gt</code>	<code>ceil</code>	<code>mldivide</code>	<code>stairs</code>
<code>permute</code>	<code>sort</code>	<code>round</code>	<code>diff</code>	<code>area</code>
<code>reshape</code>	<code>sortrows</code>	<code>min</code>	<code>sum</code>	<code>mesh</code>
<code>transpose</code>	<code>issorted</code>	<code>max</code>	<code>char</code>	<code>surf</code>
<code>ctranspose</code>		<code>mean</code>	<code>string</code>	<code>surface</code>
<code>linspace</code>		<code>median</code>	<code>cellstr</code>	
		<code>mode</code>		<code>semilogx</code>
				<code>semilogy</code>
				<code>fill</code>
				<code>fill3</code>
				<code>line</code>
				<code>text</code>

Convert Between Datetime Arrays, Numbers, and Text

In this section...

- “Overview” on page 7-42
- “Convert Between Datetime and Character Vectors” on page 7-42
- “Convert Between Datetime and String Arrays” on page 7-44
- “Convert Between Datetime and Date Vectors” on page 7-44
- “Convert Serial Date Numbers to Datetime” on page 7-45
- “Convert Datetime Arrays to Numeric Values” on page 7-45

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 7-42 — A character vector.
Example: Thursday, August 23, 2012 9:45:44.946 AM
- Date Vector on page 7-44 — 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 7-45 — A single number equal to the number of days since January 0, 0000 in the proleptic ISO calendar (specifying use of the Gregorian 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.

For a complete list of date and time format specifiers, see the **Format** property of the **datetime** data type.

```
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 = ["24-Oct-2016 11:58:17";
        "19-Nov-2016 09:36:29";
        "12-Dec-2016 10:09:06")  
  
str =  
  
3×1 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 =
1x4 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 =
```

```
1x4 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` | `duration` | `string`

More About

- “Represent Dates and Times in MATLAB” on page 7-2
- “Extract or Assign Date and Time Components of Datetime Array” on page 7-23
- Proleptic Gregorian Calendar

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

Note The best practice is to use `datetime` values to represent points in time rather than date vectors. Unlike date vectors, `datetime` values display in a human-readable format, often avoiding the need for conversion to text. If you need to convert a date vector to text, the best practice is to first convert it to a `datetime` value, and then to convert the `datetime` value to text by using the `string` or `char` functions. While you can convert date vectors to text directly by using the `datestr` function, you might get unexpected results, as described in this section.

Because a date vector is a 1-by-6 row vector of numbers, the `datestr` function might interpret input date vectors as vectors of serial date numbers and return unexpected output. Or it might interpret vectors of serial date numbers as date vectors. This ambiguity exists because `datestr` has a heuristic rule for interpreting a 1-by-6 row vector as either a date vector or a vector of six serial date numbers. The same ambiguity applies to inputs that are m -by-6 numeric matrices, where each row can be interpreted either as a date vector or as six serial date numbers.

For example, 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.

```
d = datestr([3000 11 05 10 32 56])
d =
6×11 char array
'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 text '**18-Mar-0008**' (the date that is 3000 days after 0-Jan-0000). Also, `datestr` converts the next five elements as though they also were serial date numbers.

There are two methods for converting such a date vector to text.

- The **recommended** method is to convert the date vector to a `datetime` value. Then convert it using the `char`, `cellstr`, or `string` function. The `datetime` function always treats 1-by-6 numeric vectors as date vectors.

```
dt = datetime([3000 11 05 10 32 56]);
ds = string(dt)
dt =
```

```
"05-Nov-3000 10:32:56"
```

- As an alternative, convert it to a serial date number using the `datenum` function. 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 text, `datestr` interprets input as either date vectors or serial date numbers using a heuristic rule. Consider an m -by-6 matrix. The `datestr` function 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 an m -by-6 matrix of 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 2020:

```
d = datestr([2020 06 21 10 51 00])
d =
'21-Jun-2020 10:51:00'
```

But given a day value outside the typical range (1–31), `datestr` returns a date for each element of the vector.

```
d = datestr([2020 06 2110 10 51 00])
d =
6×11 char array
'12-Jul-0005'
'06-Jan-0000'
'10-Oct-0005'
'10-Jan-0000'
'20-Feb-0000'
'00-Jan-0000'
```

Again, the `datetime` function always treats numeric inputs as date vectors. In this case, it calculates an appropriate date, interpreting 2110 as the 2110th day since the beginning of June 2020.

```
d = datetime([2020 06 2110 10 51 00])
d =
datetime
11-Mar-2026 10:51:00
```

- When you have a matrix of date vectors that `datestr` might interpret incorrectly as serial date numbers, convert the matrix by using either the `datetime` or `datenum` functions. Then convert those values to text.
- 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.

See Also

`char` | `datenum` | `datestr` | `datetime` | `datevec` | `string`

More About

- “Represent Dates and Times in MATLAB” on page 7-2
- “Convert Between Datetime Arrays, Numbers, and Text” on page 7-42

Categorical Arrays

- “Create Categorical Arrays” on page 8-2
- “Convert Text in Table Variables to Categorical” on page 8-6
- “Plot Categorical Data” on page 8-10
- “Compare Categorical Array Elements” on page 8-16
- “Combine Categorical Arrays” on page 8-19
- “Combine Categorical Arrays Using Multiplication” on page 8-22
- “Access Data Using Categorical Arrays” on page 8-24
- “Work with Protected Categorical Arrays” on page 8-30
- “Advantages of Using Categorical Arrays” on page 8-34
- “Ordinal Categorical Arrays” on page 8-36
- “Core Functions Supporting Categorical Arrays” on page 8-39

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
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
{'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
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
    {'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 = ["Earth", "Jupiter", "Neptune", "Jupiter", "Mars", "Earth"]
str = 1x6 string
    "Earth"      "Jupiter"      "Neptune"      "Jupiter"      "Mars"      "Earth"
```

Convert `str` to a categorical array.

```
planets = categorical(str)
planets = 1x6 categorical
    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
    Columns 1 through 6
        "Earth"      "Jupiter"      "Neptune"      "Jupiter"      "Mars"      "Earth"
    Columns 7 through 8
        <missing>      "Mars"
```

```
planets = categorical(str)
```

```
planets = 1x8 categorical
    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 8-6
- “Access Data Using Categorical Arrays” on page 8-24
- “Compare Categorical Array Elements” on page 8-16

More About

- “Advantages of Using Categorical Arrays” on page 8-34
- “Ordinal Categorical Arrays” on page 8-36

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. The same workflow applies for table variables that are string arrays.

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	11412	cell	
Height	100x1	800	double	
LastName	100x1	11616	cell	
Location	100x1	14208	cell	
SelfAssessedHealthStatus	100x1	11540	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)
```

	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)
```

	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 Tables” on page 9-34
- “Access Data Using Categorical Arrays” on page 8-24

More About

- “Advantages of Using Categorical Arrays” on page 8-34
- “Ordinal Categorical Arrays” on page 8-36

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	11412	cell	
Height	100x1	800	double	
LastName	100x1	11616	cell	
Location	100x1	14208	cell	
SelfAssessedHealthStatus	100x1	11540	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`.

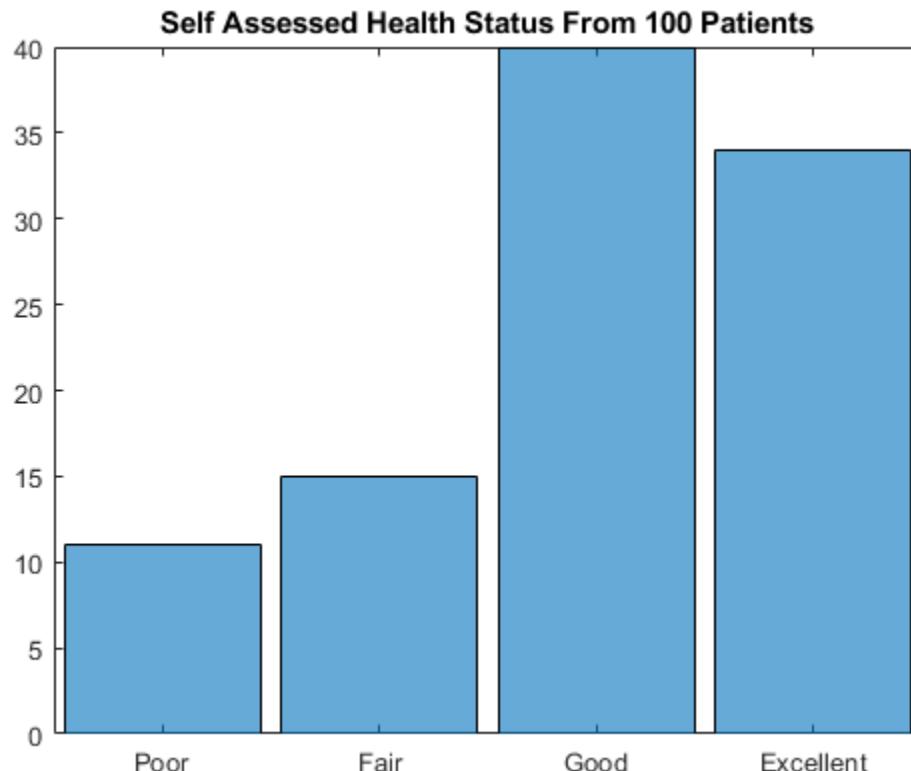
```
summary(SelfAssessedHealthStatus)
```

Poor	11
Fair	15
Good	40
Excellent	34

Plot Histogram

Create a histogram bar plot directly from a categorical array.

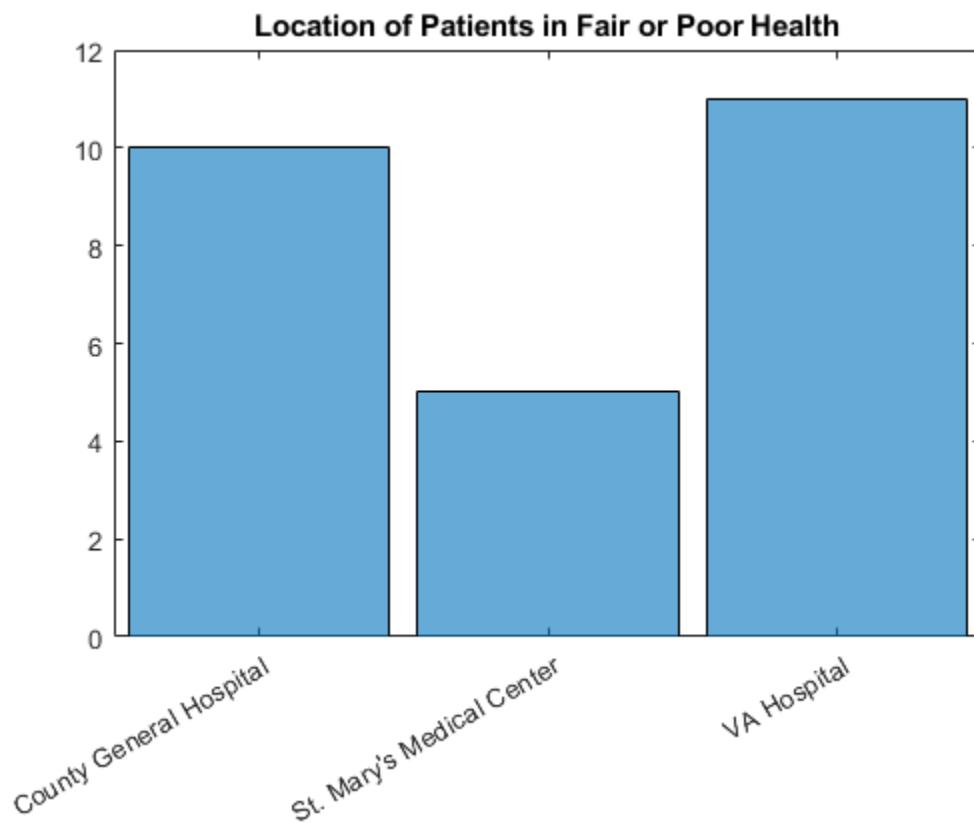
```
figure  
histogram(SelfAssessedHealthStatus)  
title('Self Assessed Health Status From 100 Patients')
```



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.

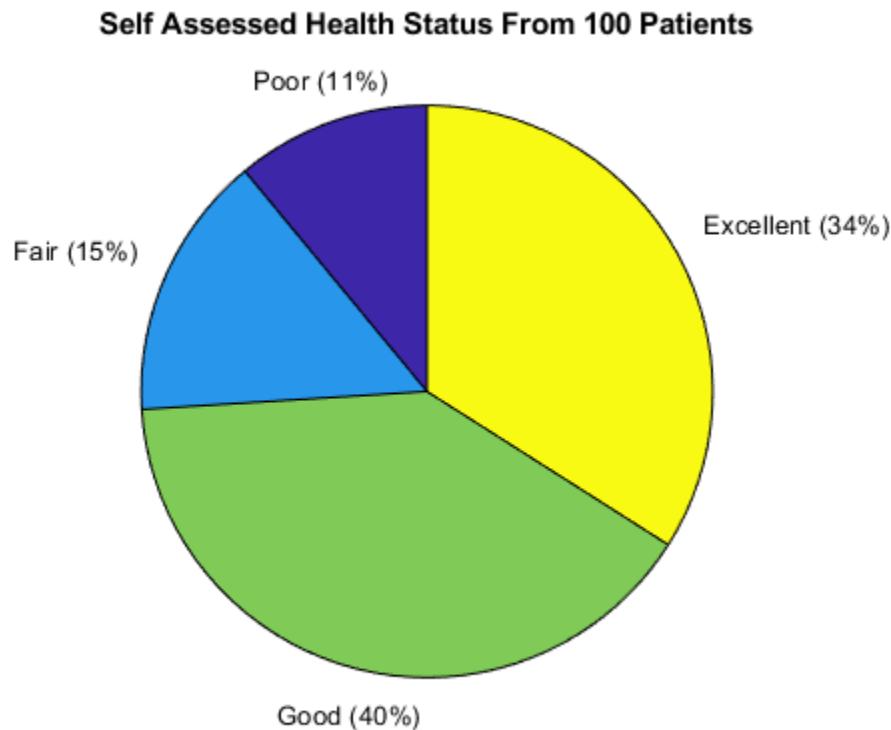
```
figure  
histogram(Location(SelfAssessedHealthStatus<='Fair'))  
title('Location of Patients in Fair or Poor Health')
```



Create Pie Chart

Create a pie chart directly from a categorical array.

```
figure  
pie(SelfAssessedHealthStatus);  
title('Self Assessed Health Status From 100 Patients')
```

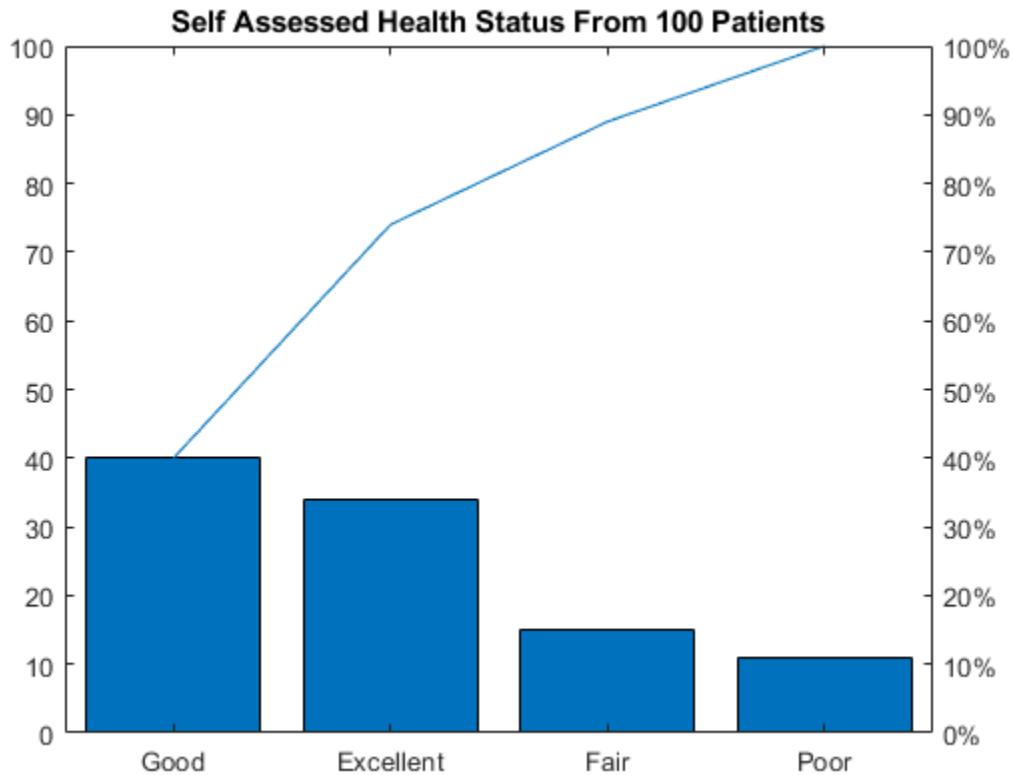


The function `pie` accepts the categorical array, `SelfAssessedHealthStatus`, and plots a pie chart of the four categories.

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');
```

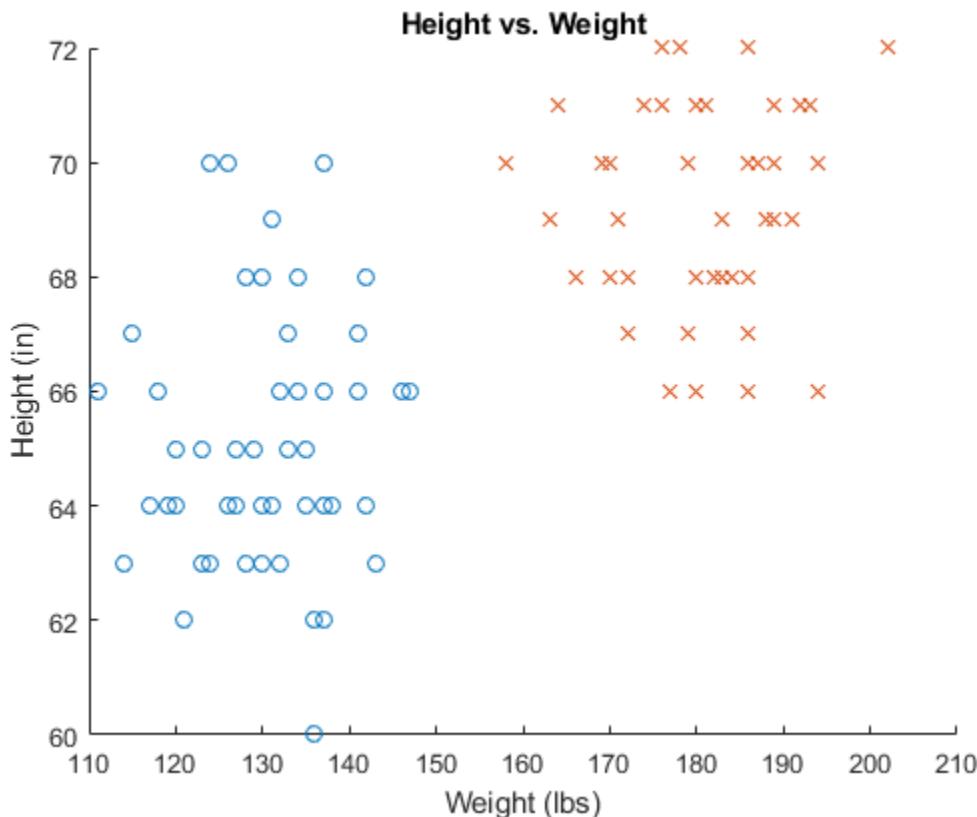
`X1` and `Y1` are 53-by-1 numeric arrays containing data from the female patients.

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 8-24

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  
    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  
    {'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
    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
  {'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 8-24

More About

- “Relational Operations”
- “Advantages of Using Categorical Arrays” on page 8-34
- “Ordinal Categorical Arrays” on page 8-36

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.

```
rng('default')
A = randi(3,[25,1]);
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	6
water	5
juice	14

Six students in classroom A prefer milk, five prefer water, and fourteen prefer juice.

Create another categorical array, B, containing the preferences of 28 students in classroom B.

```
B = randi(3,[28,1]);
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	9
water	8
juice	11

Nine students in classroom B prefer milk, eight prefer water, and eleven 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	15
water	13
juice	25

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 = randi(4,[50,1]);
Group2 = categorical(Group2,1:4,['juice' 'milk' 'soda' 'water']);
```

Summarize the categorical array, Group2.

```
summary(Group2)
```

juice	12
milk	14
soda	10
water	14

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	29
water	27
juice	37
soda	10

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
  {'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
  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 8-22
- “Convert Text in Table Variables to Categorical” on page 8-6
- “Access Data Using Categorical Arrays” on page 8-24

More About

- “Ordinal Categorical Arrays” on page 8-36

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
    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
  {'blue +' }
  {'blue -' }
  {'green +' }
  {'green -' }
  {'red +' }
  {'red -' }
```

As a consequence, `A.*B` does not equal `B.*A`.

```
D = B.*A
```

```
D = 1x3 categorical
    + blue      - red      + green
```

```
categories(D)
```

```
ans = 6x1 cell
  {'+ 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
    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
  {'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 8-19
- “Access Data Using Categorical Arrays” on page 8-24

More About

- “Ordinal Categorical Arrays” on page 8-36

Access Data Using Categorical Arrays

In this section...

["Select Data By Category" on page 8-24](#)

["Common Ways to Access Data Using Categorical Arrays" on page 8-24](#)

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	11412	cell	
Height	100x1	800	double	
LastName	100x1	11616	cell	
Location	100x1	14208	cell	
SelfAssessedHealthStatus	100x1	11540	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`.

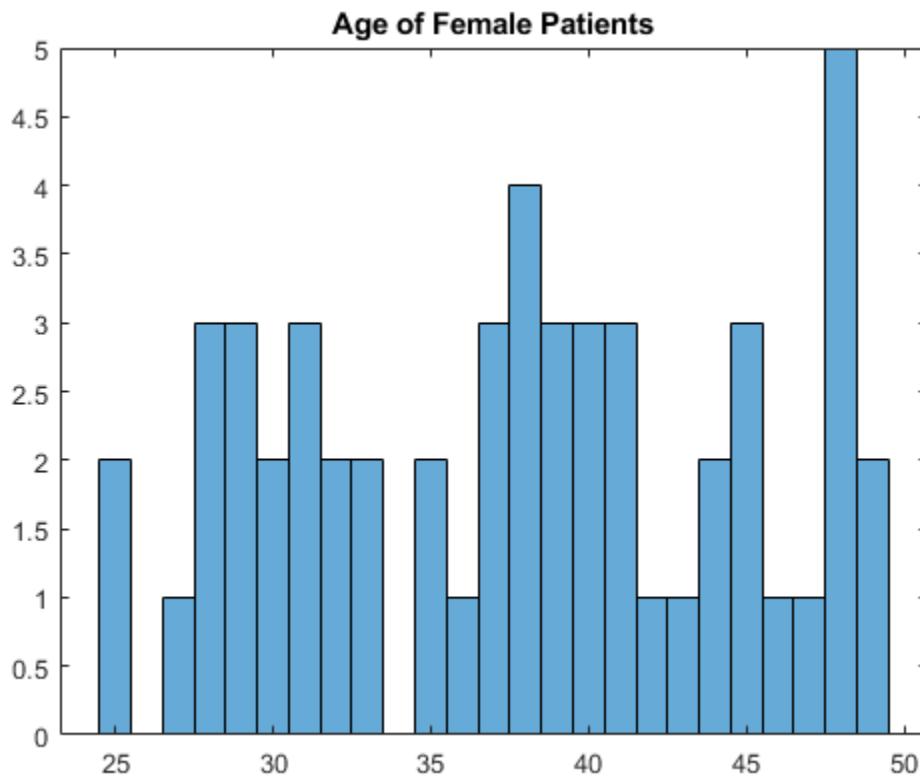
```
summary(Gender)
```

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
{'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
    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 8-6
- “Plot Categorical Data” on page 8-10
- “Compare Categorical Array Elements” on page 8-16
- “Work with Protected Categorical Arrays” on page 8-30

More About

- “Advantages of Using Categorical Arrays” on page 8-34
- “Ordinal Categorical Arrays” on page 8-36

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
    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
    {'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
    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  
    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  
    {'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  
    {'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  
    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  
{'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 8-6
- “Access Data Using Categorical Arrays” on page 8-24
- “Combine Categorical Arrays” on page 8-19
- “Combine Categorical Arrays Using Multiplication” on page 8-22

More About

- “Ordinal Categorical Arrays” on page 8-36

Advantages of Using Categorical Arrays

In this section...

- “Natural Representation of Categorical Data” on page 8-34
- “Mathematical Ordering for Character Vectors” on page 8-34
- “Reduce Memory Requirements” on page 8-34

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	16200	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
{'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	476	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 8-6
- “Compare Categorical Array Elements” on page 8-16
- “Access Data Using Categorical Arrays” on page 8-24

More About

- “Ordinal Categorical Arrays” on page 8-36

Ordinal Categorical Arrays

In this section...

["Order of Categories" on page 8-36](#)

["How to Create Ordinal Categorical Arrays" on page 8-36](#)

["Working with Ordinal Categorical Arrays" on page 8-38](#)

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
    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
    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
    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 8-30.

See Also

`categorical` | `categories` | `isequal` | `isordinal`

Related Examples

- “Create Categorical Arrays” on page 8-2
- “Convert Text in Table Variables to Categorical” on page 8-6
- “Compare Categorical Array Elements” on page 8-16
- “Access Data Using Categorical Arrays” on page 8-24

More About

- “Advantages of Using Categorical Arrays” on page 8-34

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.

<code>size</code>	<code>isequal</code>	<code>intersect</code>	<code>plot</code>	<code>double</code>
<code>length</code>	<code>isequaln</code>	<code>ismember</code>	<code>plot3</code>	<code>single</code>
<code>ndims</code>		<code>setdiff</code>	<code>scatter</code>	<code>int8</code>
<code>numel</code>	<code>eq</code>	<code>setxor</code>	<code>scatter3</code>	<code>int16</code>
<code>isrow</code>	<code>ne</code>	<code>unique</code>	<code>bar</code>	<code>int32</code>
<code>iscolumn</code>	<code>lt</code>	<code>union</code>	<code>barmh</code>	<code>int64</code>
<code>cat</code>	<code>le</code>	<code>times</code>	<code>histogram</code>	<code>uint8</code>
<code>horzcat</code>	<code>ge</code>		<code>pie</code>	<code>uint16</code>
<code>vertcat</code>	<code>gt</code>	<code>sort</code>	<code>rose</code>	<code>uint32</code>
	<code>min</code>	<code>sortrows</code>	<code>stem</code>	<code>uint64</code>
	<code>max</code>	<code>issorted</code>	<code>stairs</code>	<code>char</code>
	<code>median</code>	<code>permute</code>	<code>area</code>	<code>string</code>
	<code>mode</code>	<code>reshape</code>	<code>mesh</code>	
		<code>transpose</code>	<code>surf</code>	
		<code>ctranspose</code>	<code>surface</code>	
			<code>semilogx</code>	
			<code>semilogy</code>	
			<code>fill</code>	
			<code>fill3</code>	
			<code>line</code>	
			<code>text</code>	<code>cellstr</code>

Tables

- “Create and Work with Tables” on page 9-2
- “Add and Delete Table Rows” on page 9-11
- “Add, Delete, and Rearrange Table Variables” on page 9-14
- “Clean Messy and Missing Data in Tables” on page 9-21
- “Modify Units, Descriptions, and Table Variable Names” on page 9-26
- “Add Custom Properties to Tables and Timetables” on page 9-29
- “Access Data in Tables” on page 9-34
- “Calculations on Tables” on page 9-47
- “Split Data into Groups and Calculate Statistics” on page 9-51
- “Split Table Data Variables and Apply Functions” on page 9-54
- “Advantages of Using Tables” on page 9-58
- “Grouping Variables To Split Data” on page 9-63
- “Changes to DimensionNames Property in R2016b” on page 9-66

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	11412	cell	
Height	100x1	800	double	
LastName	100x1	11616	cell	
Location	100x1	14208	cell	
SelfAssessedHealthStatus	100x1	11540	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=5×4 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=5×10 table
 LastName      Gender      Age      Location      Height      Weight      Smo
 _____
 {'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=5×5 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=5×5 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=5×4 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=5×5 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 = 1×2

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=2×5 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';
```

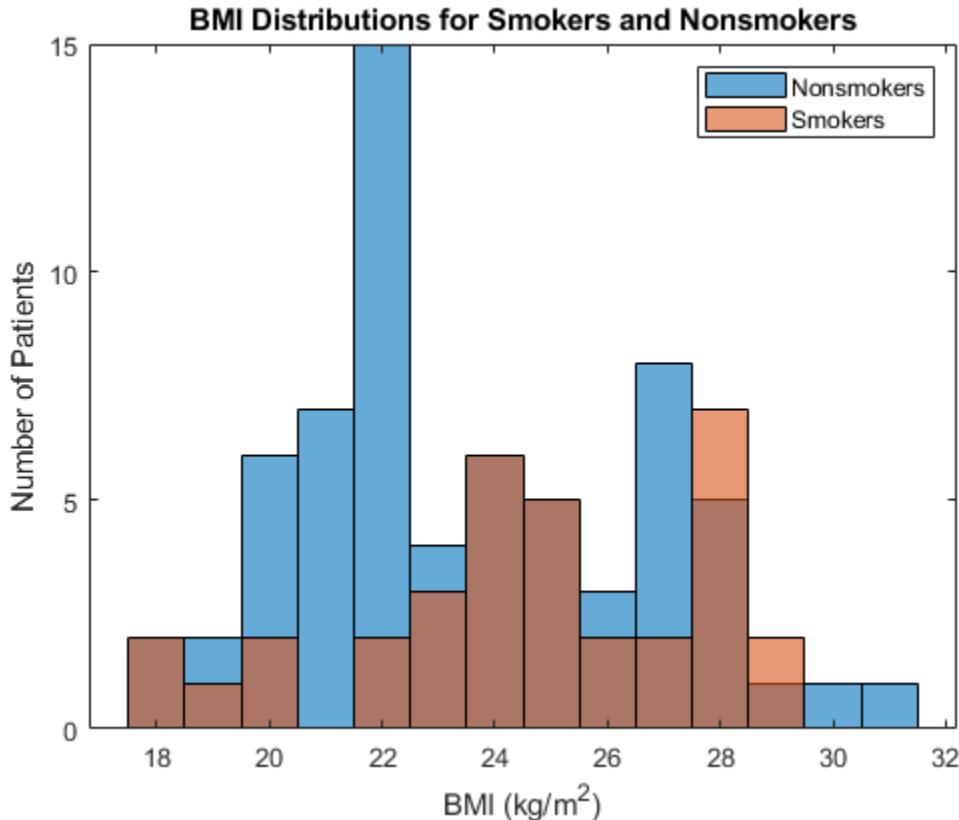
Create a histogram to explore whether there is a relationship between smoking and body-mass-index in this group of patients. You can index into BMI with the logical values from the `Smoker` table variable, because each row contains BMI and Smoker values for the same patient.

```
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;

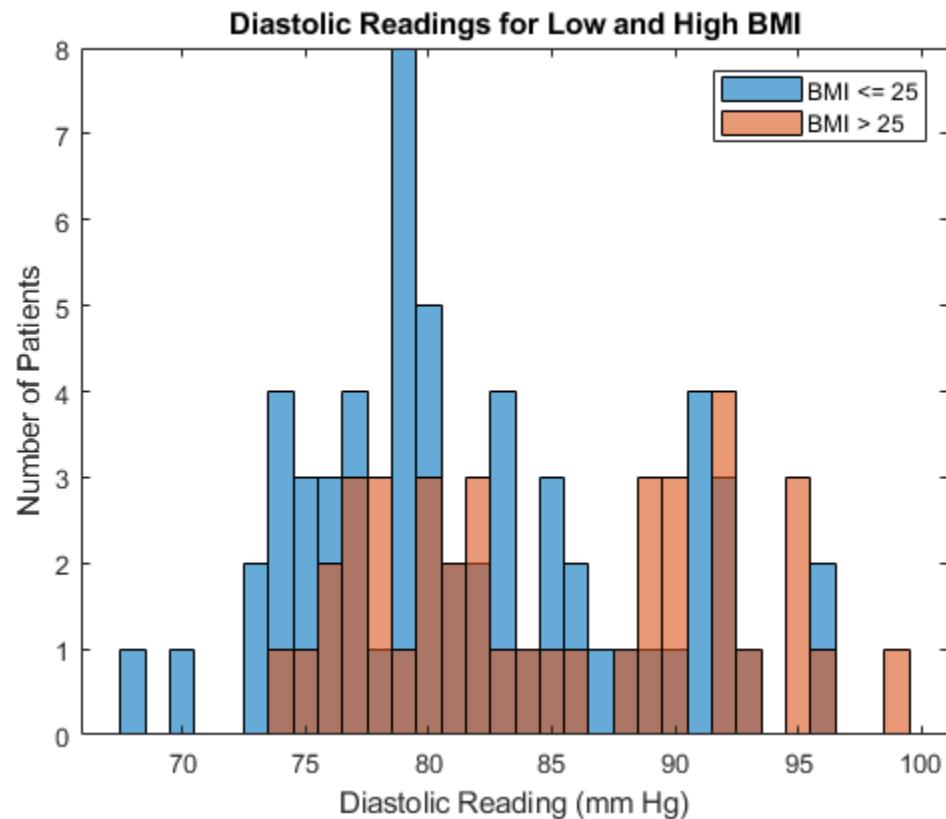
```

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=5×8 table
```

	Gender	Smoker	Height	Weight	ID	BMI	Systolic	Diastolic
Adams	{'Female'}	false	66	137	8235	22.112	127	83
Alexander	{'Male'}	true	69	171	1300	25.252	128	99
Allen	{'Female'}	false	63	143	7432	25.331	113	80
Anderson	{'Female'}	false	68	128	1577	19.462	114	77
Bailey	{'Female'}	false	68	130	2239	19.766	113	81

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=5×7 table
```

	Gender	Smoker	Height	Weight	ID	BMI	BloodPressure
Adams	{'Female'}	false	66	137	8235	22.112	127 83
Alexander	{'Male' }	true	69	171	1300	25.252	128 99
Allen	{'Female'}	false	63	143	7432	25.331	113 80
Anderson	{'Female'}	false	68	128	1577	19.462	114 77
Bailey	{'Female'}	false	68	130	2239	19.766	113 81

To put ID as the first column, reorder the table variables by position.

```
T = T(:,[5 1:4 6 7]);
```

```
T(1:5,:)
```

```
ans=5×7 table
```

	ID	Gender	Smoker	Height	Weight	BMI	BloodPressure
Adams	8235	{'Female'}	false	66	137	22.112	127 83
Alexander	1300	{'Male' }	true	69	171	25.252	128 99
Allen	7432	{'Female'}	false	63	143	25.331	113 80
Anderson	1577	{'Female'}	false	68	128	19.462	114 77
Bailey	2239	{'Female'}	false	68	130	19.766	113 81

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 9-21
- “Modify Units, Descriptions, and Table Variable Names” on page 9-26
- “Access Data in Tables” on page 9-34

More About

- “Advantages of Using Tables” on page 9-58

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 = 1x2
```

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 = 1x2
```

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 = 1x2
```

103 8

The table contains information on 103 patients now.

Delete Rows by Row Name

First, specify the variable of identifiers, `LastNames`, as row names. Then, delete the variable, `LastNames`, from `Tnew`. Finally, use the row name to index and delete rows.

```
Tnew.Properties.RowNames = Tnew.LastName;
Tnew.LastName = [];
Tnew('Smith', :) = [];
size(Tnew)
```

ans = 1x2

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 = 1x2

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 9-14
- “Clean Messy and Missing Data in Tables” on page 9-21

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	11412	cell	
Height	100x1	800	double	
LastName	100x1	11616	cell	
Location	100x1	14208	cell	
SelfAssessedHealthStatus	100x1	11540	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=5×7 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=5×8 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)
```

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

`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 `.`² in the calculation of `BMI` indicate element-wise division and exponentiation, respectively.

Display the first five rows of the table `T`.

```
head(T,5)
```

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)
```

LastName	Age	Gender	Smoker	Height	Weight	BMI	Systolic	Diastolic
{'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

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)
```

LastName	Age	Gender	Smoker	Height	Weight	BMI	BloodPressure
{'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	80

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)
```

LastName	Age	Gender	Smoker	Height	Weight	BMI	BloodPressure
{'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	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)
```

LastName	Gender	Smoker	Height	Weight	BMI	BloodPressure
{'Smith'}	{'Male'}	true	71	176	24.547	124
{'Johnson'}	{'Male'}	false	69	163	24.071	109
{'Williams'}	{'Female'}	false	64	131	22.486	125
{'Jones'}	{'Female'}	false	67	133	20.831	117
{'Brown'}	{'Female'}	false	64	119	20.426	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)
```

LastName	Smoker	Height	Weight	BMI	BloodPressure
{'Smith'}	true	71	176	24.547	124
{'Johnson'}	false	69	163	24.071	109
{'Williams'}	false	64	131	22.486	125
{'Jones'}	false	67	133	20.831	117
{'Brown'}	false	64	119	20.426	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)
```

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=5×6 table

LastName	Smoker	Height	Weight	BMI	BP
{'Smith'}	true	71	176	24.547	124
{'Johnson'}	false	69	163	24.071	109
{'Williams'}	false	64	131	22.486	125
{'Jones'}	false	67	133	20.831	117
{'Brown'}	false	64	119	20.426	122

Reorient Rows To Become Variables

You can reorient the rows of a table or timetable, so that they become the variables in 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=6×5 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 = 6×1

```
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 9-11
- “Clean Messy and Missing Data in Tables” on page 9-21
- “Modify Units, Descriptions, and Table Variable Names” on page 9-26

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. (Use the `disp` function to display all 21 rows, even when running this example as a live script.)

```
T = readtable('messy.csv', 'TreatAsEmpty', {'.', 'NA'});
disp(T)
```

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	{0x0 char}	563	563
{'pnj5'}	463	{'no'}	463	463
{'wnn3'}	6	{'no'}	6	6
{'oks9'}	23	{'yes'}	23	23
{'wba3'}	NaN	{'yes'}	NaN	14
{'pkn4'}	2	{'no'}	2	2
{'adw3'}	22	{'no'}	22	22
{'poj2'}	-99	{'yes'}	-99	-99
{'bas8'}	23	{'no'}	23	23
{'gry5'}	NaN	{'yes'}	NaN	21

`T` is a table with 21 rows and five variables. `'TreatAsEmpty'` only applies to numeric columns in the file and cannot handle numeric values specified as text, 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'});  
rowsWithMissing = T(any(TF,2),:);  
disp(rowsWithMissing)
```

A	B	C	D	E
{'egh3'}	NaN	{'no'}	7	7
{'abk6'}	563	{0x0 char}	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);
disp(T)
```

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	{0x0 char}	563	563
{'pnj5'}	463	{'no'}	463	463
{'wnn3'}	6	{'no'}	6	6
{'oks9'}	23	{'yes'}	23	23
{'wba3'}	NaN	{'yes'}	NaN	14
{'pkn4'}	2	{'no'}	2	2
{'adw3'}	22	{'no'}	22	22
{'poj2'}	NaN	{'yes'}	NaN	NaN
{'bas8'}	23	{'no'}	23	23
{'gry5'}	NaN	{'yes'}	NaN	21

`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');
disp(T2)
```

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
{'pkn4'}	2	{'no'}	2	2
{'adw3'}	22	{'no'}	22	22
{'poj2'}	22	{'yes'}	22	22

{'bas8'}	23	{'no'}	23	23
{'gry5'}	23	{'yes'}	23	21

Remove Rows with Missing Values

Create a new table, T3, that contains only the rows from T without missing values. T3 has only 16 rows.

```
T3 = rmmissing(T);
disp(T3)
```

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'});
disp(T3)
```

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

{'egh3'}	3	{'no'}	7	7
{'pkn4'}	2	{'no'}	2	2
{'ple2'}	2	{'no'}	2	2
{'pnj5'}	463	{'no'}	463	463
{'wnn3'}	6	{'no'}	6	6

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'});
disp(T3)
```

A	C	B	D	E
{'abk6'}	{'yes'}	563	563	563
{'afe1'}	{'yes'}	3	3	3
{'argl'}	{'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
{'egh3'}	{'no'}	3	7	7
{'pkn4'}	{'no'}	2	2	2
{'ple2'}	{'no'}	2	2	2
{'pnj5'}	{'no'}	463	463	463
{'wnn3'}	{'no'}	6	6	6

See Also

`fillmissing` | `ismissing` | `readtable` | `rmmissing` | `sortrows` | `standardizeMissing` | `summary`

Related Examples

- “Add and Delete Table Rows” on page 9-11
- “Add, Delete, and Rearrange Table Variables” on page 9-14
- “Modify Units, Descriptions, and Table Variable Names” on page 9-26
- “Access Data in Tables” on page 9-34
- “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=5×6 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:

	Column 1	Column 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=5×6 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 9-14
- “Access Data in Tables” on page 9-34

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)
```

Region	OutageTime	Loss	Customers	RestorationTime	Cause
{'SouthWest'}	2002-02-01 12:18	458.98	1.8202e+06	2002-02-07 16:50	{'winter storm'}
{'MidWest'}	2002-03-05 17:53	96.563	2.8666e+05	2002-03-10 14:41	{'wind'}
{'MidWest'}	2002-03-16 06:18	186.44	2.1275e+05	2002-03-18 23:23	{'severe storm'}

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:

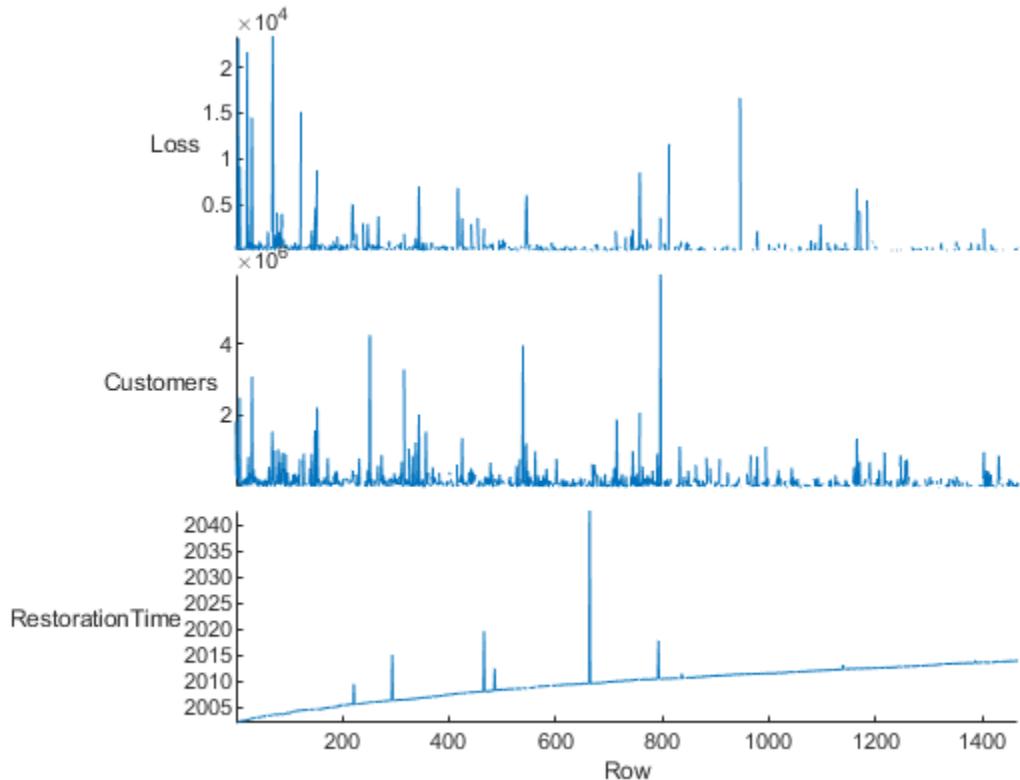
    Description: ''
    UserData: []
    DimensionNames: {'Row'  'Variables'}
    VariableNames: {1x6 cell}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: []
    RowNames: {}

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 9-26
- “Access Data in Tables” on page 9-34
- “Add, Delete, and Rearrange Table Variables” on page 9-14

Access Data in Tables

In this section...

- "Summary of Table Indexing Syntaxes" on page 9-34
- "Tables Containing Specified Rows and Variables" on page 9-37
- "Extract Data Using Dot Notation and Logical Values" on page 9-40
- "Dot Notation with Any Variable Name or Expression" on page 9-42
- "Extract Data from Specified Rows and Variables" on page 9-44

A table is a container that stores column-oriented data in variables. Table variables can have different data types and sizes as long as all variables have the same number of rows. Table variables have names, just as the fields of a structure have names. The rows of a table can have names, but row names are not required. To access table data, index into the rows and variables using either their names or numeric indices.

Typical reasons for indexing into tables include:

- Reordering or removing rows and variables.
- Adding arrays as new rows or variables.
- Extracting arrays of data to use as input arguments to functions.

Summary of Table Indexing Syntaxes

Depending on the type of indexing you use, you can access either a subtable or an array extracted from the table. Indexing with:

- **Smooth parentheses, ()**, returns a table that has selected rows and variables.
- **Dot notation** returns the contents of a variable as an array.
- **Curly braces, {}**, returns an array concatenated from the contents of selected rows and variables.

You can specify rows and variables by name, numeric index, or data type. Starting in R2019b, variable names and row names can include any characters, including spaces and non-ASCII characters. Also, they can start with any characters, not just letters. Variable and row names do not have to be valid MATLAB identifiers (as determined by the `isvarname` function).

Type of Output	Syntax	Rows	Variables	Examples
Table, containing specified rows and variables	<code>T(rows, vars)</code>	<p>Specified as:</p> <ul style="list-style-type: none"> • Row numbers (between 1 and m) • Names, if T has row names • Times, if T is a timetable • Colon (:), meaning all rows 	<p>Specified as:</p> <ul style="list-style-type: none"> • Variable numbers (between 1 and n) • Names • Colon (:), meaning all variables 	<ul style="list-style-type: none"> • <code>T(1:5, [1 4 5])</code> Table having the first five rows and the first, fourth, and fifth variables of T • <code>T(:, {'A', 'B'})</code> Table having all rows and the variables named 'A' and 'B' from T
Table, containing variables that have specified data type	<code>S = vartype(type); T(rows, S)</code>	<p>Specified as:</p> <ul style="list-style-type: none"> • Row numbers (between 1 and m) • Names, if T has row names • Times, if T is a timetable • Colon (:), meaning all rows 	Specified as a data type, such as 'numeric', 'categorical', or 'datetime'	<ul style="list-style-type: none"> • <code>S = vartype('numeric');</code> <code>T(1:5, S)</code> Table having the first five rows and the numeric variables of T
Array, extracting data from one variable	<code>T.var</code> <code>T.(expression)</code>	Not specified	<p>Specified as:</p> <ul style="list-style-type: none"> • A variable name (without quotation marks) • An expression inside parentheses that returns a variable name or number 	<ul style="list-style-type: none"> • <code>T.Date</code> Array extracted from table variable named 'Date' • <code>T.('2019/06/30')</code> Array extracted from table variable named '2019/06/30' • <code>T.(1)</code> Array extracted from the first table variable

Type of Output	Syntax	Rows	Variables	Examples
Array, extracting data from one variable and specified rows	<code>T.var(rows)</code> <code>T.(expression)(rows)</code>	Specified as numeric or logical indices of the array	Specified as: <ul style="list-style-type: none">• A variable name (without quotation marks)• An expression inside parentheses that returns a variable name or number	<ul style="list-style-type: none">• <code>T.Date(1:5)</code> First five rows of array extracted from table variable named 'Date'• <code>T.('2019/06/30')(1:5)</code> First five rows of array extracted from table variable named '2019/06/30'• <code>T.(1)(1:5)</code> First five rows of array extracted from the first table variable
Array, concatenating data from specified rows and variables	<code>T{rows,vars}</code>	Specified as: <ul style="list-style-type: none">• Row numbers (between 1 and m)• Names, if T has row names• Times, if T is a timetable• Colon (:), meaning all rows	Specified as: <ul style="list-style-type: none">• Variable numbers (between 1 and n)• Names• Colon (:), meaning all variables	<ul style="list-style-type: none">• <code>T{1:5,[1 4 5]}</code> Array concatenated from the first five rows and the first, fourth, and fifth variables of T• <code>T{:, {'A','B'}}</code> Array concatenated from all rows and the variables named 'A' and 'B' from T

Type of Output	Syntax	Rows	Variables	Examples
Array, concatenating data from specified rows and variables with specified data type	<code>S = vartype(type); T{rows,S}</code>	Specified as: <ul style="list-style-type: none"> • Row numbers (between 1 and m) • Names, if T has row names • Times, if T is a timetable • Colon (:), meaning all rows 	Specified as a data type, such as 'numeric', 'categorical', or 'datetime'	<ul style="list-style-type: none"> • <code>S = vartype('numeric');</code> <code>T{1:5,S}</code> Array concatenated from the first five rows and the numeric variables of T
Array, concatenating data from all rows and variables	<code>T.Variables</code>	Not specified	Not specified	<ul style="list-style-type: none"> • <code>T.Variables</code> Identical to array returned by <code>T{:, :}</code>

Tables Containing Specified Rows and Variables

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	11412	cell	
Height	100x1	800	double	
LastName	100x1	11616	cell	
Location	100x1	14208	cell	
SelfAssessedHealthStatus	100x1	11540	cell	
Smoker	100x1	100	logical	
Systolic	100x1	800	double	
Weight	100x1	800	double	

Create a table and populate it with the Age, Gender, Height, Weight, and Smoker workspace variables. Use the unique identifiers in LastName as row names. T is a 100-by-5 table. (When you specify row names, they do not count as a table variable).

```
T = table(Age,Gender,Height,Weight,Smoker, ...
'RowNames',LastName)
```

T=100x5 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
Davis	46	{'Female'}	68	142	false
Miller	33	{'Female'}	64	142	true
Wilson	40	{'Male'}	68	180	false
Moore	28	{'Male'}	68	183	false
Taylor	31	{'Female'}	66	132	false
Anderson	45	{'Female'}	68	128	false
Thomas	42	{'Female'}	66	137	false
Jackson	25	{'Male'}	71	174	false
White	39	{'Male'}	72	202	true
Harris	36	{'Female'}	65	129	false
Martin	48	{'Male'}	71	181	true
:					

Index Using Numeric Indices

Create a subtable containing the first five rows and all the variables from T. To specify the desired rows and variables, use numeric indices within parentheses. This type of indexing is similar to indexing into numeric arrays.

```
T1 = T(1:5,:)
```

T1=5×5 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 'Williams' and 'Brown'. Since T has row names that are the last names of patients, index into T using row names.

```
T2 = T({'Williams', 'Brown'},:)
```

T2=2×5 table

	Age	Gender	Height	Weight	Smoker
Williams	38	{'Female'}	64	131	false
Brown	49	{'Female'}	64	119	false

T2 is a 2-by-5 table.

You also can select variables by name. Create a table that has only the first five rows of T and the Height and Weight variables. Display it.

```
T3 = T(1:5,['Height','Weight'])
```

T3=5×2 table

	Height	Weight
Smith	71	176
Johnson	69	163
Williams	64	131
Jones	67	133
Brown	64	119

Table variable names do not have to be valid MATLAB identifiers. They can include spaces and non-ASCII characters, and can start with any character.

Add a variable name with spaces and a dash to T. Then index into T using variable names.

```
T = addvars(T,'SelfAssessedHealthStatus','NewVariableNames','Self-Assessed Health Status');
T(1:5,['Age','Smoker','Self-Assessed Health Status'])
```

ans=5×3 table

	Age	Smoker	Self-Assessed Health Status
Smith	38	true	{'Excellent'}
Johnson	43	false	{'Fair'}
Williams	38	false	{'Good'}
Jones	40	false	{'Fair'}
Brown	49	false	{'Good'}

Specify Data Type Subscript

Instead of specifying variables using names or numbers, you can create a data type subscript that matches all variables having the same data type.

First, create a data type subscript to match numeric table variables.

```
S = vartype('numeric')

S =
  table vartype subscript:
    Select table variables matching the type 'numeric'
```

See Access Data in a Table.

Create a table that has only the numeric variables, and only the first five rows, from T.

```
T4 = T(1:5,S)
```

T4=5×3 table

	Age	Height	Weight
Smith	38	71	176
Johnson	43	69	163

Williams	38	64	131
Jones	40	67	133
Brown	49	64	119

Extract Data Using Dot Notation and Logical Values

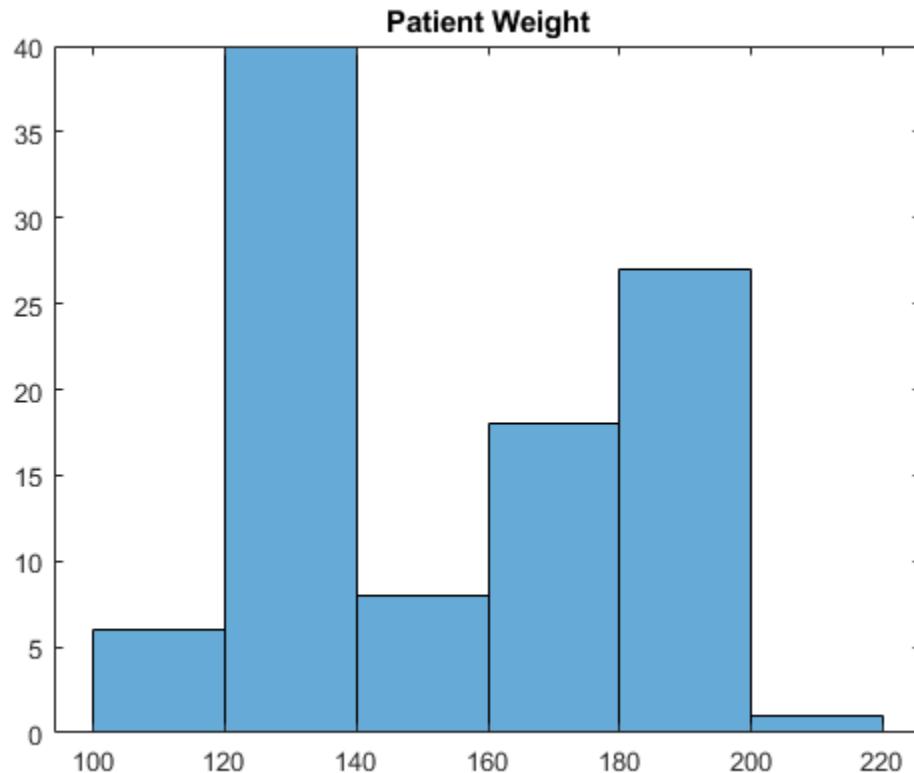
Create a table from the `patients` MAT-file. Then use dot notation to extract data from table variables. You can also index using logical indices generated from values in a table variable that meet a condition.

```
load patients  
T = table(Age,Gender,Height,Weight,Smoker,...  
'RowNames',LastName);
```

Extract Data from Variable

To extract data from one variable, use dot notation. Extract numeric values from the variable `Weight`. Then plot a histogram of those values.

```
histogram(T.Weight)  
title('Patient Weight')
```



`T.Weight` is a double-precision column vector with 100 rows.

Select Rows with Logical Indexing

You can index into an array or a table using an array of logical indices. Typically, you use a logical expression that determines which values in a table variable meet a condition. The result of the expression is an array of logical indices.

For example, create logical indices matching patients whose age is less than 40.

```
rows = T.Age < 40
rows = 100x1 logical array
```

```
1
0
1
0
0
0
1
0
1
1
:
:
```

To extract heights for patients whose age is less than 40, index into the `Height` variable using `rows`. There are 56 patients younger than 40.

```
T.Height(rows)
```

```
ans = 56x1
```

```
71
64
64
68
66
71
72
65
69
69
:
:
```

You can index into a table with logical indices. Display the rows of `T` for the patients who are younger than 40.

```
T(rows,:)
```

```
ans=56x5 table
```

	Age	Gender	Height	Weight	Smoker
Smith	38	{'Male'}	71	176	true
Williams	38	{'Female'}	64	131	false
Miller	33	{'Female'}	64	142	true
Moore	28	{'Male'}	68	183	false

Taylor	31	{'Female'}	66	132	false
Jackson	25	{'Male'}	71	174	false
White	39	{'Male'}	72	202	true
Harris	36	{'Female'}	65	129	false
Thompson	32	{'Male'}	69	191	true
Garcia	27	{'Female'}	69	131	true
Martinez	37	{'Male'}	70	179	false
Rodriguez	39	{'Female'}	64	117	false
Walker	28	{'Female'}	65	123	true
Hall	25	{'Male'}	70	189	false
Allen	39	{'Female'}	63	143	false
Young	25	{'Female'}	63	114	false
:					

You can match multiple conditions with one logical expression. Display the rows for smoking patients younger than 40.

```
rows = (T.Smoker==true & T.Age<40);
T(rows,:)
```

ans=18x5 table					
	Age	Gender	Height	Weight	Smoker
Smith	38	{'Male'}	71	176	true
Miller	33	{'Female'}	64	142	true
White	39	{'Male'}	72	202	true
Thompson	32	{'Male'}	69	191	true
Garcia	27	{'Female'}	69	131	true
Walker	28	{'Female'}	65	123	true
King	30	{'Male'}	67	186	true
Nelson	33	{'Male'}	66	180	true
Mitchell	39	{'Male'}	71	164	true
Turner	37	{'Male'}	70	194	true
Sanders	33	{'Female'}	67	115	true
Price	31	{'Male'}	72	178	true
Jenkins	28	{'Male'}	69	189	true
Long	39	{'Male'}	68	182	true
Patterson	37	{'Female'}	65	120	true
Flores	31	{'Female'}	66	141	true
:					

Dot Notation with Any Variable Name or Expression

When you index using dot notation, there are two ways to specify a variable.

- By name, without quotation marks. For example, `T.Date` specifies a variable named 'Date'.
- By an expression, where the expression is enclosed by parentheses after the dot. For example, `T.('Start Date')` specifies a variable named 'Start Date'.

Use the first syntax when a table variable name also happens to be a valid MATLAB® identifier. (A valid identifier starts with a letter and includes only letters, digits, and underscores.)

Use the second syntax when you specify:

- A number that indicates the position of the variable in the table.
- A variable name that isn't a valid MATLAB identifier.
- A function whose output is the name of a variable in the table, or a variable you add to the table.
The output of the function must be a character vector or a string scalar.

For example, create a table from the `patients` MAT-file. Then use dot notation to access the contents of table variables.

```
load patients  
  
T = table(Age,Gender,Height,Weight,Smoker,...  
    'RowNames',LastName);
```

To specify a variable by position in the table, use a number. `Age` is the first variable in `T`, so use the number `1` to specify its position.

```
T.(1)  
  
ans = 100x1  
  
38  
43  
38  
40  
49  
46  
33  
40  
28  
31  
:  
:
```

To specify a variable by name, you can enclose it in quotation marks. Since '`Age`' is a valid identifier, you can specify it using either `T.Age` or `T.('Age')`.

```
T.( 'Age' )  
  
ans = 100x1  
  
38  
43  
38  
40  
49  
46  
33  
40  
28  
31  
:  
:
```

You can specify table variable names that are not valid MATLAB identifiers. Variable names can include spaces and non-ASCII characters, and can start with any character. However, when you use dot notation to access a table variable with such a name, you must specify it using parentheses.

Add a variable name with spaces and a hyphen to T.

```
T = addvars(T,SelfAssessedHealthStatus,'NewVariableNames','Self-Assessed Health Status');
T(1:5,:)
```

ans=5x6 table

	Age	Gender	Height	Weight	Smoker	Self-Assessed Health Status
Smith	38	{'Male'}	71	176	true	{'Excellent'}
Johnson	43	{'Male'}	69	163	false	{'Fair'}
Williams	38	{'Female'}	64	131	false	{'Good'}
Jones	40	{'Female'}	67	133	false	{'Fair'}
Brown	49	{'Female'}	64	119	false	{'Good'}

Access the new table variable using dot notation. Display the first five elements.

```
C = T.('Self-Assessed Health Status');
C(1:5)
```

```
ans = 5x1 cell
{'Excellent'}
{'Fair'}
{'Good'}
{'Fair'}
{'Good'}
```

You also can use the output of a function as a variable name. Delete the T.('Self-Assessed Health Status') variable. Then replace it with a variable whose name includes today's date.

```
T.('Self-Assessed Health Status') = [];
T.(string(datetime('today')) + ' Self Report') = SelfAssessedHealthStatus;
T(1:5,:)
```

ans=5x6 table

	Age	Gender	Height	Weight	Smoker	24-Feb-2021 Self Report
Smith	38	{'Male'}	71	176	true	{'Excellent'}
Johnson	43	{'Male'}	69	163	false	{'Fair'}
Williams	38	{'Female'}	64	131	false	{'Good'}
Jones	40	{'Female'}	67	133	false	{'Fair'}
Brown	49	{'Female'}	64	119	false	{'Good'}

Extract Data from Specified Rows and Variables

Indexing with curly braces extracts data from a table and results in an array, *not* a subtable. But other than that difference, you can specify rows and variables using numbers, names, and data type subscripts, just as you can when you index using smooth parentheses. To extract values from a table, use curly braces. If you extract values from multiple table variables, then the variables must have data types that allow them to be concatenated together.

Specify Rows and Variables

Create a table from numeric and logical arrays from the `patients` file.

```
load patients

T = table(Age,Height,Weight,Smoker, ...
    'RowNames', LastName);
```

Extract data from multiple variables in `T`. Unlike dot notation, indexing with curly braces can extract values from multiple table variables and concatenate them into one array.

Extract the height and weight for the first five patients. Use numeric indices to select the first five rows, and variable names to select the variables `Height` and `Weight`.

```
A = T{1:5,['Height','Weight']}
```

```
A = 5×2
```

71	176
69	163
64	131
67	133
64	119

`A` is a 5-by-2 numeric array, not a table.

If you specify one variable name, then curly brace indexing results in the same array you can get with dot notation. However, you must specify both rows and variables when you use curly brace indexing. For example, this syntaxes `T.Height` and `T{:, 'Height'}` return the same array.

Extract Data from All Rows and Variables

If all the table variables have data types that allow them to be concatenated together, then you can use the `T.Variables` syntax to put all the table data into an array. This syntax is equivalent to `T{:, :}` where the colons indicate all rows and all variables.

```
A2 = T.Variables
```

```
A2 = 100×4
```

38	71	176	1
43	69	163	0
38	64	131	0
40	67	133	0
49	64	119	0
46	68	142	0
33	64	142	1
40	68	180	0
28	68	183	0
31	66	132	0
:			

See Also

`addvars` | `histogram` | `table` | `vartype`

Related Examples

- “Advantages of Using Tables” on page 9-58
- “Create and Work with Tables” on page 9-2
- “Modify Units, Descriptions, and Table Variable Names” on page 9-26
- “Calculations on Tables” on page 9-47
- “Find Array Elements That Meet a Condition” on page 5-2

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. Starting in R2018a, you also can use the `groupsummary` function for calculations on groups of data in a table.

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=10x4 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. First, compute the means by using the varfun function.

```
varfun(@mean,T,'InputVariables','TestAvg',...
    'GroupingVariables','Gender')
```

Gender	GroupCount	mean_TestAvg
{'female'}	5	87.067
{'male'}	5	83.4

Starting in R2018a, you also can use the `groupsummary` function to perform computations on groups of data in a table. Compute the maximum values of `TestAvg` for each group of students using `groupsummary`.

```
groupsummary(T, 'Gender', 'max', 'TestAvg')
```

Gender	GroupCount	max_TestAvg
{'female'}	5	95.333
{'male'}	5	90

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
```

	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'
```

	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](#) | [groupsummary](#) | [rowfun](#) | [splitapply](#) | [summary](#) | [table](#) | [varfun](#)

Related Examples

- “Access Data in Tables” on page 9-34
- “Split Table Data Variables and Apply Functions” on page 9-54

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	330	categorical	
Height	100x1	800	double	
LastName	100x1	11616	cell	
Location	100x1	14208	cell	
SelfAssessedHealthStatus	100x1	560	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)
```

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 = 4x1

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=4x7 table						
gender	smoker	numPatients	numPF	stdDiastolicPF	stdDiastolicGE	BMI
Female	false	40	10	6.8872	3.9012	21.672
Female	true	13	5	5.4129	5.0409	21.669
Male	false	26	8	4.2678	4.8159	26.578
Male	true	21	3	5.6862	5.258	26.458

See Also

[findgroups](#) | [splitapply](#)

Related Examples

- “Split Table Data Variables and Apply Functions” on page 9-54

More About

- “Grouping Variables To Split Data” on page 9-63

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,:)
```

Region	OutageTime	Loss	Customers	RestorationTime	Cause
SouthWest	2002-02-01 12:18	458.98	1.8202e+06	2002-02-07 16:50	winter storm
SouthEast	2003-01-23 00:49	530.14	2.1204e+05	NaT	winter storm
SouthEast	2003-02-07 21:15	289.4	1.4294e+05	2003-02-17 08:14	winter storm
West	2004-04-06 05:44	434.81	3.4037e+05	2004-04-06 06:10	equipment fault
MidWest	2002-03-16 06:18	186.44	2.1275e+05	2002-03-18 23:23	severe storm

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=10x2 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=15×4 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=15×5 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 Tables” on page 9-34
- “Calculations on Tables” on page 9-47
- “Split Data into Groups and Calculate Statistics” on page 9-51

More About

- “Grouping Variables To Split Data” on page 9-63

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	330	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,:)
```

	Gender	Age	Smoker	BloodPressure
Smith	Male	38	true	124
Johnson	Male	43	false	109
Williams	Female	38	false	125
Jones	Female	40	false	117
Brown	Female	49	false	122

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'})
```

	Age	BloodPressure
Williams	38	125
Brown	49	122

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)
```

	Age	BloodPressure
Williams	38	125
Brown	49	122

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 Tables” on page 9-34.

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:
Column 1     Column 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 9-26
- “Access Data in Tables” on page 9-34

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.

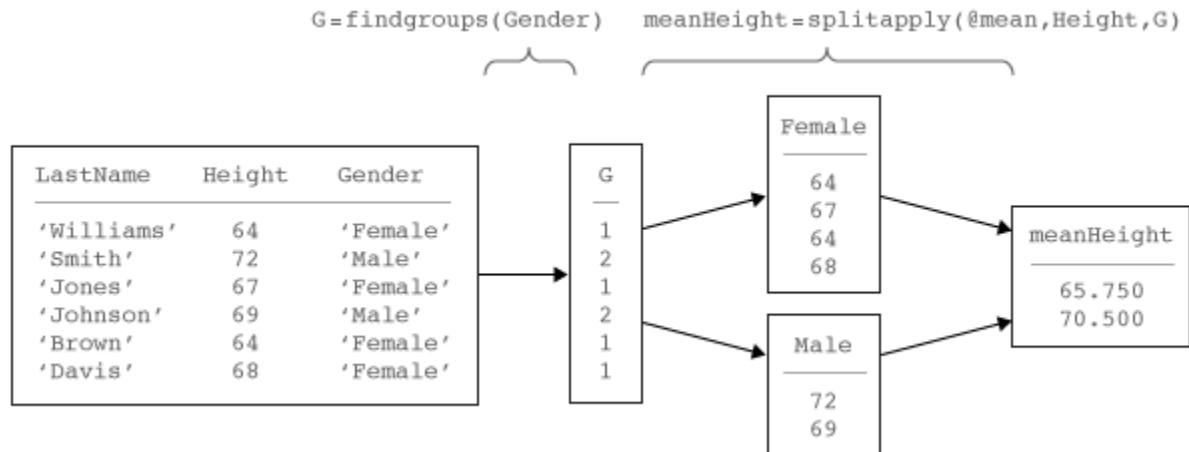
The `findgroups` function can accept multiple grouping variables, for example `G = findgroups(A1,A2)`. You also can include multiple grouping variables in a table, for example `T = table(A1,A2); G = findgroups(T)`. The `findgroups` function defines groups by the unique combinations of values across corresponding 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	<code>NaN</code>
Logical	(Cannot be missing)
Categorical	<code><undefined></code>
<code>datetime</code>	<code>NaT</code>
<code>duration</code>	<code>NaN</code>
Cell array of character vectors	<code>''</code>
String	<code><missing></code>

See Also

`findgroups` | `rowfun` | `splitapply` | `varfun`

Related Examples

- “Access Data in Tables” on page 9-34
- “Split Table Data Variables and Apply Functions” on page 9-54
- “Split Data into Groups and Calculate Statistics” on page 9-51

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 10-5
- “Combine Timetables and Synchronize Their Data” on page 10-8
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-14
- “Select Times in Timetable” on page 10-19
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27
- “Using Row Labels in Table and Timetable Operations” on page 10-36
- “Loma Prieta Earthquake Analysis” on page 10-41
- “Preprocess and Explore Time-stamped Data Using timetable” on page 10-51

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 `readtimetable` function. The output is a timetable.

```
indoors = readtimetable('indoors.csv');
```

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=5×2 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=5×3 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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:24	36	80	49	51.3
2015-11-15 01:13:35	36	80	NaN	NaN
2015-11-15 01:30:24	NaN	NaN	48.9	51.5
2015-11-15 02:26:47	37	79	NaN	NaN
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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:24	36	80	49	51.3
2015-11-15 01:13:35	36	80	48.919	51.463
2015-11-15 01:30:24	36.23	79.77	48.9	51.5
2015-11-15 02:26:47	37	79	48.9	51.5
2015-11-15 03:00:24	37	80.378	48.9	51.5

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.

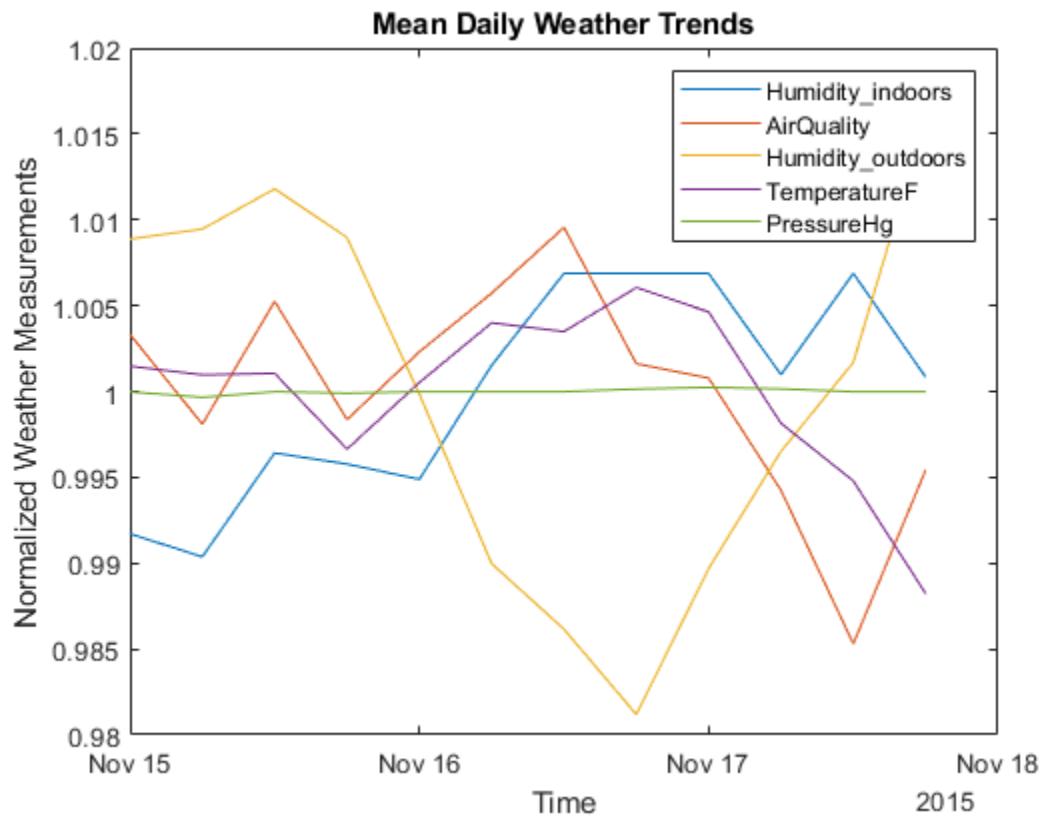
```
tv = [datetime(2015,11,15):hours(6):datetime(2015,11,18)];
ttHourly = retime(ttLinear, tv, 'mean');
ttHourly = rmmissing(ttHourly);
```

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 10-5
- “Combine Timetables and Synchronize Their Data” on page 10-8
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-14
- “Select Times in Timetable” on page 10-19
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27

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 `retime` 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=5×3 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
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=5×3 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,:)
```

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
```

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,:)
```

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=5×3 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 10-8
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-14
- “Select Times in Timetable” on page 10-19
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27

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
```

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
```

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]
```

Time	AAPL	FB
08-Jan-2016 09:00:00	98.55	99.88
08-Jan-2016 09:00:00	98.55	99.88

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=5×2 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 `opOtherStocks`. The output timetable has one set of row times and the variables from both timetables.

`op = [opWeek1 opOtherStocks]`

op=5×4 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,:)
```

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,:)
```

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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:24	36	80	49	51.3
2015-11-15 01:13:35	36	80	NaN	NaN
2015-11-15 01:30:24	NaN	NaN	48.9	51.5
2015-11-15 02:26:47	37	79	NaN	NaN
2015-11-15 03:00:24	NaN	NaN	48.9	51.5

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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:24	36	80	49	51.3
2015-11-15 01:13:35	36	80	48.919	51.463
2015-11-15 01:30:24	36.23	79.77	48.9	51.5
2015-11-15 02:26:47	37	79	48.9	51.5
2015-11-15 03:00:24	37	80.378	48.9	51.5

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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:00	36	80	49	51.299
2015-11-15 01:00:00	36	80	48.934	51.432
2015-11-15 02:00:00	36.634	79.366	48.9	51.5
2015-11-15 03:00:00	37	80.361	48.9	51.5
2015-11-15 04:00:00	36.727	81.453	48.834	51.5

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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:00	36	80	49	51.299
2015-11-15 00:30:00	36	80	48.967	51.366
2015-11-15 01:00:00	36	80	48.934	51.432
2015-11-15 01:30:00	36.224	79.776	48.9	51.499
2015-11-15 02:00:00	36.634	79.366	48.9	51.5

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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:00	36	80	49	51.299
2015-11-15 00:30:00	36	80	48.967	51.366
2015-11-15 01:00:00	36	80	48.934	51.432
2015-11-15 01:30:00	36.224	79.776	48.9	51.499
2015-11-15 02:00:00	36.634	79.366	48.9	51.5

2015-11-15 00:00:00	36	80	49	51.299
2015-11-15 00:30:00	36	80	48.967	51.366
2015-11-15 01:00:00	36	80	48.934	51.432
2015-11-15 01:30:00	36.224	79.776	48.9	51.499
2015-11-15 02:00:00	36.634	79.366	48.9	51.5

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
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:00	36.5	80.05	48.931	51.394
2015-11-16 00:00:00	36.85	80.35	47.924	51.571
2015-11-17 00:00:00	36.85	79.45	48.45	51.238
2015-11-18 00:00:00	NaN	NaN	49.5	50.8

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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:00	36.4	80.2	48.9	51.45
2015-11-15 06:00:00	36.4	79.8	48.9	51.45
2015-11-15 12:00:00	36.6	80.4	49.025	51.45
2015-11-15 18:00:00	36.6	79.8	48.9	51.225
2015-11-16 00:00:00	36.6	80.2	48.5	51.4

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,:)
```

Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF
2015-11-15 00:00:00	36.4	80.2	48.9	51.45
2015-11-15 06:00:00	36.4	79.8	48.9	51.45
2015-11-15 12:00:00	36.6	80.4	49.025	51.45
2015-11-15 18:00:00	36.6	79.8	48.9	51.225

2015-11-16 00:00:00	36.6	80.2	48.5	51.4
---------------------	------	------	------	------

See Also

`retime | synchronize | table2timetable | timetable`

Related Examples

- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-5
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-14
- “Select Times in Timetable” on page 10-19
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27

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=5×4 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=7×4 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>

	Date	Tmax	Tmin	PrecipTotal	WXEvent
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=7×4 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=7×4 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=5×1 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 10-5
- “Combine Timetables and Synchronize Their Data” on page 10-8
- “Select Times in Timetable” on page 10-19
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27

Select Times in Timetable

A timetable is a type of table that associates a time with each row. You can select time-based subsets of its data in several ways:

- Find times within a certain range using the `timerange` or `withtol` functions.
- Match recurring units of time, such as days or months, using the components of `datetime` arrays.
- Resample or group data with the `retime` function.

For example, read a sample file `outages.csv`, containing data representing electric utility outages in the United States from 2002–2014. The vector of row times, `OutageTime`, indicates when the outages occurred. The `readtimetable` function imports it as a `datetime` array. Display the first five rows.

```
TT = readtimetable('outages.csv');
head(TT,5)
```

OutageTime	Region	Loss	Customers	RestorationTime	Cause
2002-02-01 12:18	{'SouthWest'}	458.98	1.8202e+06	2002-02-07 16:50	{'winter storm'}
2003-01-23 00:49	{'SouthEast'}	530.14	2.1204e+05	NaT	{'winter storm'}
2003-02-07 21:15	{'SouthEast'}	289.4	1.4294e+05	2003-02-17 08:14	{'winter storm'}
2004-04-06 05:44	{'West'}	434.81	3.4037e+05	2004-04-06 06:10	{'equipment failure'}
2002-03-16 06:18	{'MidWest'}	186.44	2.1275e+05	2002-03-18 23:23	{'severe storm'}

Before R2019a, read tabular data with `readtable` and convert it to a timetable using `table2timetable`.

Select Time Range

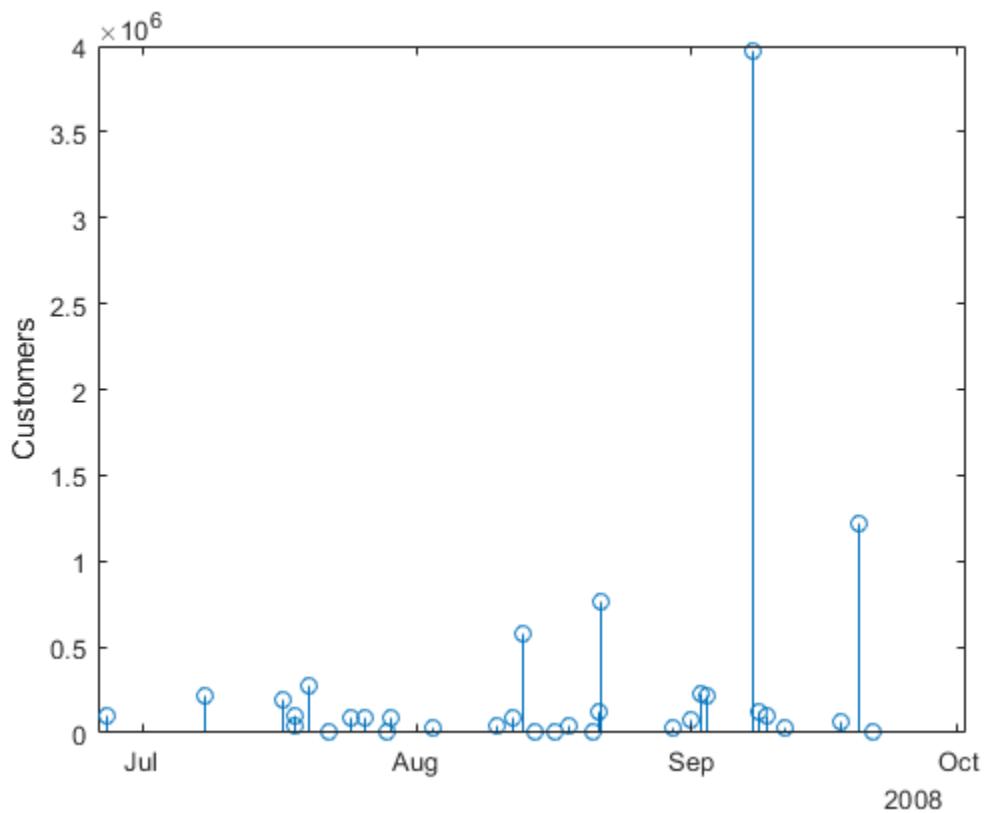
To find data in a specific range, you can use the `timerange` function, which defines time-based subscripts for indexing. For instance, define a range for the summer of 2008, which started on June 20 and ended on September 21. By default, `timerange` defines a half-open interval that is closed on the left and open on the right, so specify the end date as September 22.

```
TR = timerange("2008-06-20","2008-09-22")
TR =
    timetable timerange subscript:
    Select timetable rows with times in the half-open interval:
    [20-Jun-2008 00:00:00, 22-Sep-2008 00:00:00)
```

See [Select Timetable Data by Row Time and Variable Type](#).

Find the outages that occurred in that range, and then plot the number of customers affected over time.

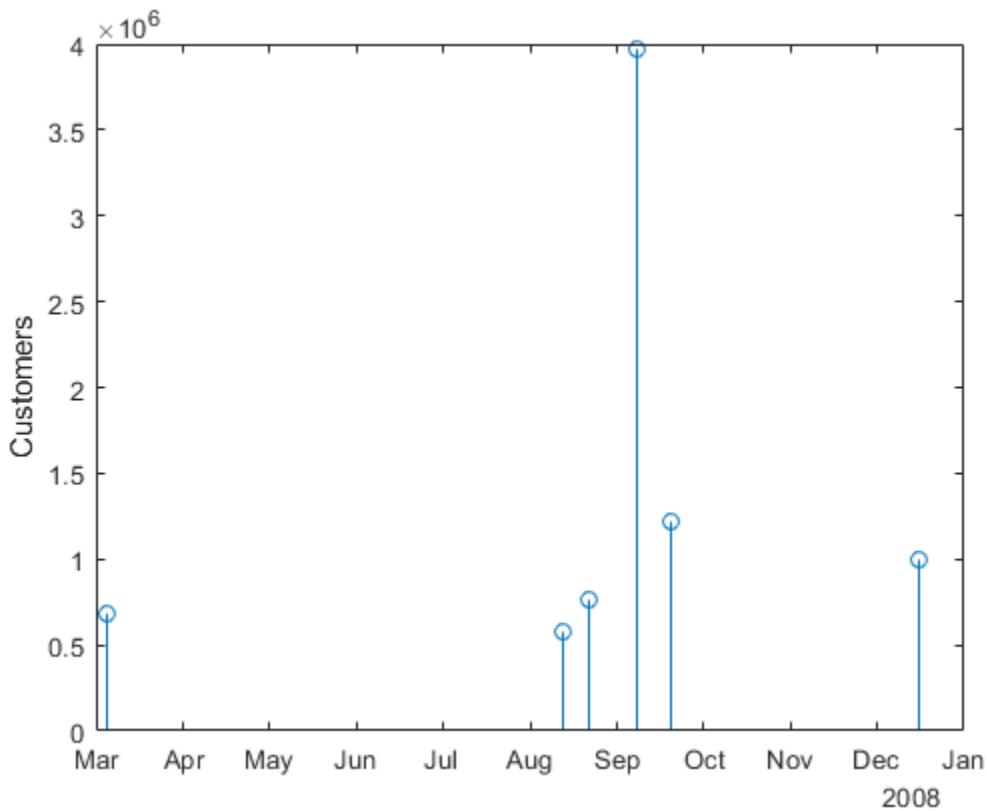
```
summer08 = TT(TR,:);
stem(summer08.OutageTime,summer08.Customers)
ylabel("Customers")
```



Several outages during that time range had high customer impact. Expand the range to a time period that spans the entire year of 2008 and look for similarly high numbers.

```
TR = timerange("2008","years");
all08 = TT(TR,:);
high08 = all08(all08.Customers > 500000,:,:);

stem(high08.OutageTime,high08.Customers)
ylabel('Customers')
```



The `timerange` function is also helpful for selecting specific dates. Selecting times by comparing `datetime` values can give misleading results because all `datetime` values include both date and time components. However, when you specify only the date component of a `datetime` value, the time component is set to midnight. Therefore, although there is data from June 26, a comparison like this one returns no results.

```
any(summer08.OutageTime == datetime("2008-06-26"))
```

```
ans = logical  
0
```

Instead, you can use `timerange`.

```
TR = timerange("2008-06-26", "days");  
june26 = summer08(TR, :)
```

OutageTime	Region	Loss	Customers	RestorationTime	Cause
2008-06-26 22:36	{'NorthEast'}	425.21	93612	2008-06-27 06:53	{'thunder storm'}

Another way to define a range is to specify a tolerance around a time using `withtol`. For example, find rows from the summer of 2008 where `OutageTime` is within three days of Labor Day, September 1.

```
WT = withtol("2008-09-01", days(3));
nearSep1 = summer08(WT,:)
```

nearSep1=4×5 timetable

OutageTime	Region	Loss	Customers	RestorationTime	Cause
2008-09-01 23:35	{'SouthEast'}	206.27	2.27e+05		NaT
2008-09-01 00:18	{'MidWest'}	510.05	74213	2008-09-01 14:07	{'thunder storm'}
2008-09-02 19:01	{'MidWest'}	NaN	2.215e+05	2008-09-03 02:58	{'severe storm'}
2008-08-29 20:25	{'West'}	NaN	31624	2008-09-01 01:51	{'wind'}

Match Units of Time

You also can use units of `datetime` values, such as hours or days, to identify rows for logical indexing. This method can be useful for specifying periodic intervals.

For example, find the values of `OutageTime` whose month components have values of 3 or less, corresponding to January, February, and March of each year. Use the resulting logical array to index into `TT`.

```
TR = (month(TT.OutageTime) <= 3);
winterTT = TT(TR,:);
```

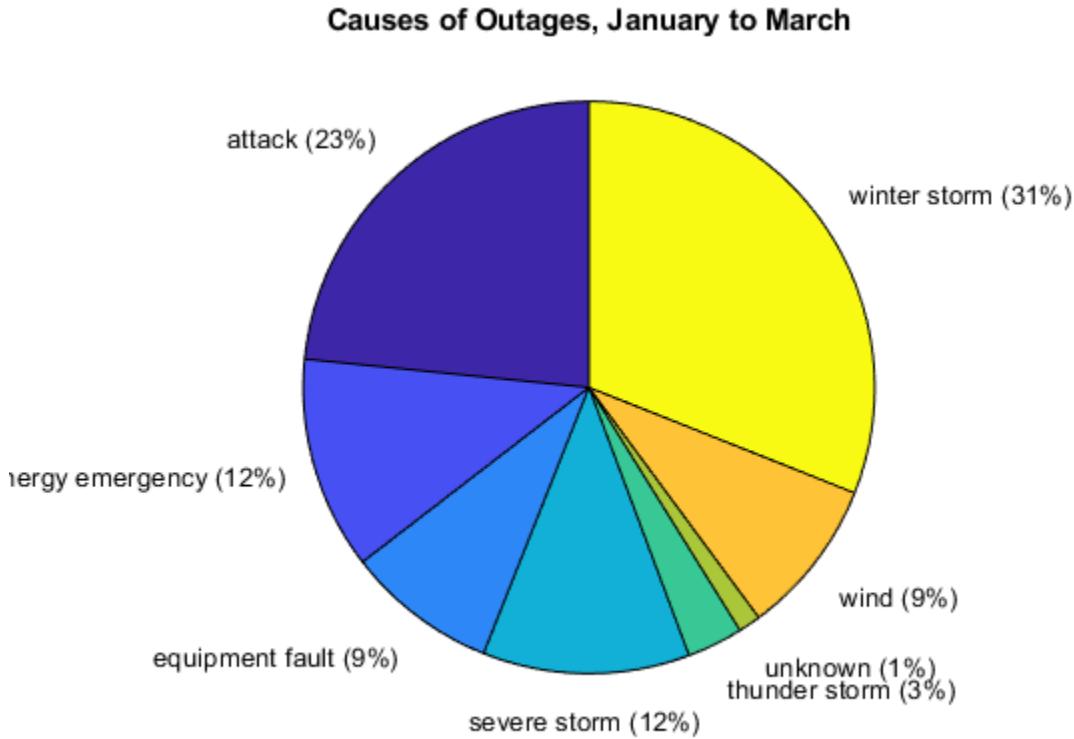
`head(winterTT,5)`

ans=5×5 timetable

OutageTime	Region	Loss	Customers	RestorationTime	Cause
2002-02-01 12:18	{'SouthWest'}	458.98	1.8202e+06	2002-02-07 16:50	{'winter storm'}
2003-01-23 00:49	{'SouthEast'}	530.14	2.1204e+05		{'winter storm'}
2003-02-07 21:15	{'SouthEast'}	289.4	1.4294e+05	2003-02-17 08:14	{'winter storm'}
2002-03-16 06:18	{'MidWest'}	186.44	2.1275e+05	2002-03-18 23:23	{'severe storm'}
2005-02-04 08:18	{'MidWest'}	NaN	NaN	2005-02-04 19:51	{'attack'}

Create a pie chart of the wintertime causes. The `pie` function accepts only numeric or `categorical` inputs, so first convert `Cause` to `categorical`.

```
winterTT.Cause = categorical(winterTT.Cause);
pie(winterTT.Cause)
title("Causes of Outages, January to March");
```



Group by Time Period

The `retime` function adjusts row times to create specified intervals, either by resampling or grouping values. Its pre-defined intervals range from seconds to years, and you can specify how to handle missing or multiple values for the intervals. For instance, you can select the first observation from each week, or count observations in a quarter.

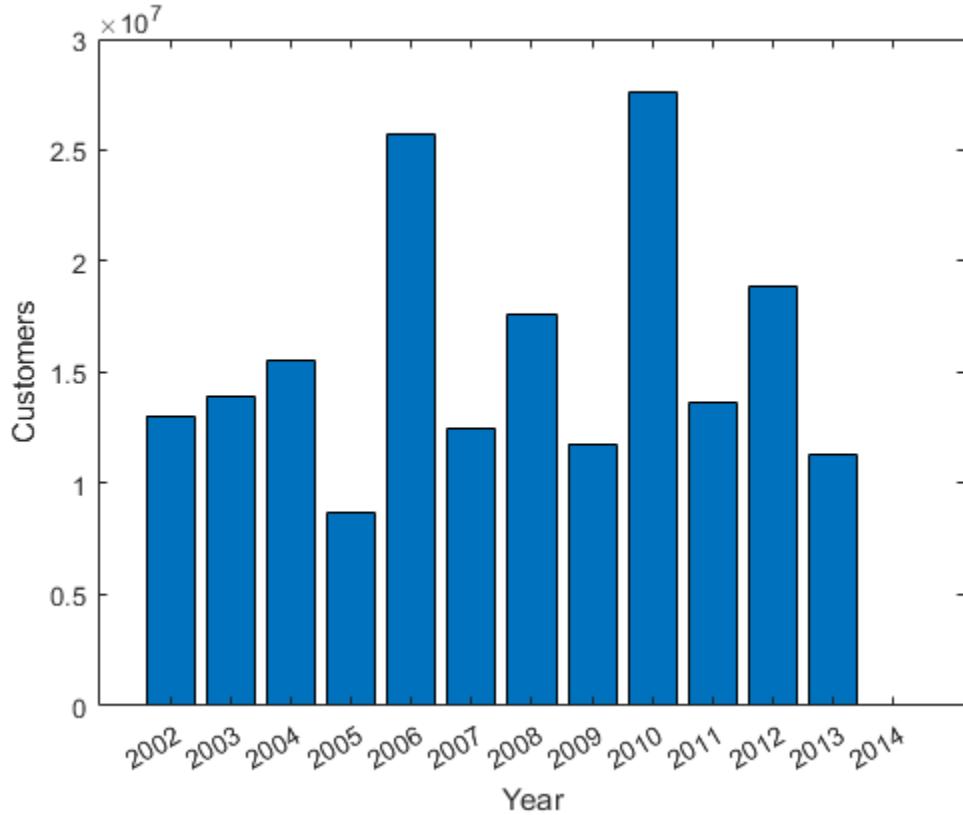
For the outage data, you can use `retime` to find totals for each year. First, create a timetable with only numeric variables. Then, call `retime` and specify a yearly interval, combining multiple values using a sum. The output has one row for each year, containing the total losses and total customers affected during that year.

```
numTT = TT(:, varType("numeric"));
numTT = retime(numTT, "yearly", "sum");
head(numTT, 5)

ans=5×2 timetable
OutageTime      Loss      Customers
_____
2002-01-01 00:00    81335    1.3052e+07
2003-01-01 00:00    58036    1.396e+07
2004-01-01 00:00    51014    1.5523e+07
2005-01-01 00:00    33980    8.7334e+06
2006-01-01 00:00    35129    2.5729e+07
```

Create a bar chart of the number of customers affected each year.

```
bar(numTT.OutageTime,numTT.Customers)
xlabel("Year")
ylabel("Customers")
```



Calculate Durations Using Row Times

You can use the row times of a timetable with other `datetime` or `duration` values to perform calculations. For example, calculate the durations of the power outages listed in the outage data. Then calculate the monthly medians of the outage durations and plot them.

First add the outage durations to TT by subtracting the row times (which are the starts of power outages) from `RestorationTime` (which are the ends of the power outages). Change the format of `OutageDuration` to display the durations of the outages in days. Display the first five rows of TT.

```
TT.OutageDuration = TT.RestorationTime - TT.OutageTime;
TT.OutageDuration.Format = 'd';
head(TT,5)
```

OutageTime	Region	Loss	Customers	RestorationTime	Cause
2002-02-01 12:18	{'SouthWest'}	458.98	1.8202e+06	2002-02-07 16:50	{'winter storm'}
2003-01-23 00:49	{'SouthEast'}	530.14	2.1204e+05	NaT	{'winter storm'}
2003-02-07 21:15	{'SouthEast'}	289.4	1.4294e+05	2003-02-17 08:14	{'winter storm'}
2004-04-06 05:44	{'West'}	434.81	3.4037e+05	2004-04-06 06:10	{'equipment failure'}
2002-03-16 06:18	{'MidWest'}	186.44	2.1275e+05	2002-03-18 23:23	{'severe storm'}

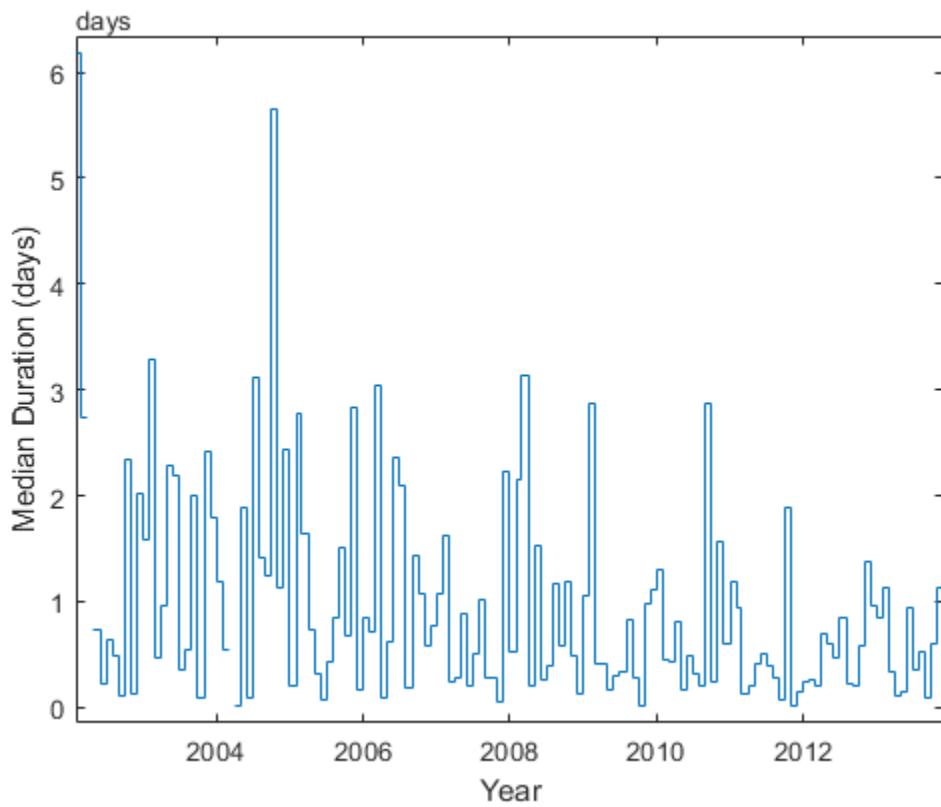
Create a timetable that has only the outage durations. Some rows of TT have missing values, `NaN`, for the restoration times, leading to `NaN` values in `OutageDuration`. To remove the `NaN` values from `medianTT`, use the `rmmissing` function. Then use `retime` to calculate the monthly median outage duration. Display the first five rows of `medianTT`.

```
medianTT = TT(:, "OutageDuration");
medianTT = rmmissing(medianTT);
medianTT = retime(medianTT, 'monthly', @median);
head(medianTT, 5)
```

OutageTime	OutageDuration
2002-02-01 00:00	6.1889 days
2002-03-01 00:00	2.7472 days
2002-04-01 00:00	NaN days
2002-05-01 00:00	0.72917 days
2002-06-01 00:00	0.22431 days

Create a staircase chart of the monthly median outage durations.

```
stairs(medianTT.OutageTime, medianTT.OutageDuration)
xlabel("Year")
ylabel("Median Duration (days)")
```



See Also

[NaT](#) | [categorical](#) | [datetime](#) | [duration](#) | [month](#) | [readtimetable](#) | [retime](#) | [rmmissing](#) | [timerange](#) | [timetable](#) | [vartype](#) | [withtol](#)

Related Examples

- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-5
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27
- “Access Data in Tables” on page 9-34
- “Calculations on Tables” on page 9-47
- “Compare Dates and Time” on page 7-33
- “Date and Time Arithmetic” on page 7-28

Clean Timetable with Missing, Duplicate, or Nonuniform Times

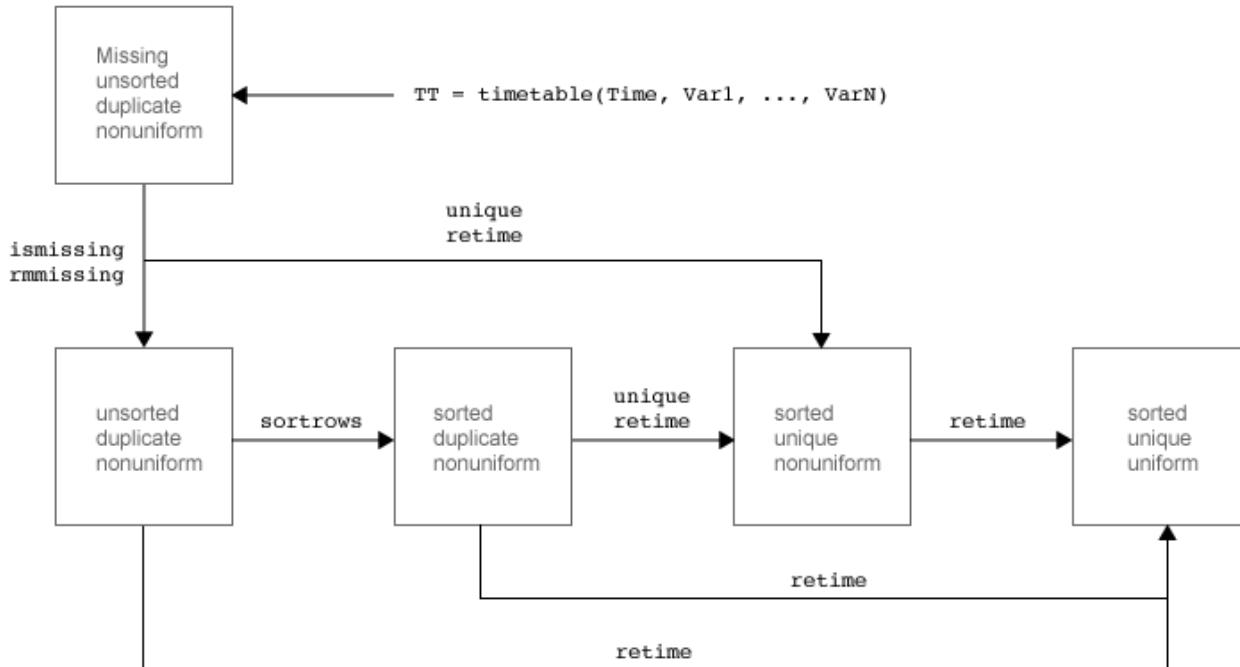
This example shows how to create a *regular* timetable from one that has missing, duplicate, or nonuniform times. A timetable is a type of table that associates a time-stamp, or *row time*, with each row of data. In a regular timetable, the row times are sorted and unique, and differ by the same regular time step.

Also, some toolboxes have functions that work on regularly spaced time series data in the form of numeric arrays. So the example also shows how to export the data from a timetable for use with other functions.

There are a number of issues with row times that can make timetables irregular. The row times can be missing. They can be out of order. They can be duplicates, creating multiple rows with the same time that might have the same or different data. And even when they are present, sorted, and unique, they can differ by time steps of different sizes.

Timetables provide a number of different ways to resolve missing, duplicate, or nonuniform times, and to resample or aggregate data to create 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 vector of unique times 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 `TT` includes temperature, rainfall, and wind speed measurements taken at irregular times during that day.

```
load badTimes
TT
```

TT=12×3 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

Find and Remove Rows with Missing Row Times

One way to begin is by finding and removing rows that have a `NaT`, or missing value, as the row time. To find missing values in the vector of row times, use `ismissing`. The `ismissing` function returns a logical vector that contains 1 wherever `TT.Time` has a missing value.

```
natRowTimes = ismissing(TT.Time)
natRowTimes = 12x1 logical array
```

0
0
0
0
1
0
0
0
0
0
0
:

To keep only those rows that do not have missing values as row times, index into `TT` using `~natRowTimes` as row indices. Assign those rows to a new timetable, `goodRowTimesTT`.

```
goodRowTimesTT = TT(~natRowTimes,:)
goodRowTimesTT=11×3 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 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

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 timetable variables still might have missing data values. For example, the last row of `goodRowTimesTT` has `NaN` values for the `Rain` and `WindSpeed` variables.

Remove Rows with Missing Times and Missing Data

As an alternative, you can remove both missing row times and missing data values at the same time by using the `rmmissing` function. `rmmissing` removes any timetable rows that have missing row times, missing data values, or both.

Display the missing row time and missing data values of `TT`.

`TT`

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 all rows that have missing row times or data values. Assign the remaining rows to the timetable `goodValuesTT`.

```
goodValuesTT = rmmissing(TT)
```

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 if It Is Regular

After dealing with missing values, you can go on to sort your timetable and then determine if the sorted timetable is regular.

To determine if `goodValuesTT` is already sorted, use the `issorted` function.

```
tf = issorted(goodValuesTT)  
tf = logical  
0
```

Since it is not, sort the timetable on its row times by using the `sortrows` function.

```
sortedTT = sortrows(goodValuesTT)  
  
sortedTT=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 `sortedTT` 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(sortedTT)  
tf = logical  
0
```

Since it is not, display the differences between row times.

```
diff(sortedTT.Time)  
  
ans = 9x1 duration  
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
```

Since the row times are sorted, this result shows that some row times are unique and some are duplicates.

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 `sortedTT` are duplicate rows. (There are other rows in `sortedTT` that have duplicate row times but differing data values.)

To remove the duplicate rows from `sortedTT`, use `unique`. The `unique` function returns the unique rows and sorts them by their row times.

```
uniqueRowsTT = unique(sortedTT)
```

uniqueRowsTT=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, `uniqueRowsTT` 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 `uniqueRowsTT`.

```
dupTimes = sort(uniqueRowsTT.Time);
tf = (diff(dupTimes) == 0);
dupTimes = dupTimes(tf);
dupTimes = unique(dupTimes)

dupTimes = 2x1 datetime
09-Jun-2016 08:49:10
09-Jun-2016 09:53:57
```

To display the rows with duplicate row times, index into `uniqueRowsTT` using `dupTimes`. When you index on times, the output timetable contains all rows with matching row times.

```
uniqueRowsTT(dupTimes, :)
```

ans=6×3 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

When a timetable has rows with duplicate times, you might want to select particular rows and discard the other rows having duplicate times. For example, you can select either the first or the last of the rows with duplicate row times by using the `unique` and `retime` functions.

First, create a vector of unique row times from `TT` by using `unique`.

```
uniqueTimes = unique(uniqueRowsTT.Time)

uniqueTimes = 5x1 datetime
09-Jun-2016 05:03:11
09-Jun-2016 06:01:04
09-Jun-2016 07:59:23
09-Jun-2016 08:49:10
09-Jun-2016 09:53:57
```

Select the first row from each set of rows that have duplicate times. To copy data from the first rows, specify the '`firstvalue`' method.

```
firstUniqueRowsTT = retime(uniqueRowsTT,uniqueTimes,'firstvalue')
```

firstUniqueRowsTT=5×3 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. To copy data from the last rows, specify the '`lastvalue`' method.

```
lastUniqueRowsTT = retime(uniqueRowsTT,uniqueTimes,'lastvalue')
```

lastUniqueRowsTT=5×3 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

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

As a result, the last two rows of `firstUniqueRowsTT` and `lastUniqueRowsTT` have different values in the `Temp` variable.

Aggregate Data from All Rows with Duplicate Times

Another way to deal with data in the rows having duplicate times is to aggregate or combine the data values in some way. 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.

```
meanTT = retime(uniqueRowsTT,uniqueTimes,'mean')
```

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

As a result, the last two rows of `meanTT` have mean temperatures in the `Temp` variable for the rows with duplicate row times.

Make Timetable Regular

Finally, you can resample data from an irregular timetable to make it regular by using the `retime` function. For example, you can interpolate the data from `meanTT` onto a regular hourly time vector. To use linear interpolation, specify '`linear`'. Each row time in `hourlyTT` begins on the hour, and there is a one-hour interval between consecutive row times.

```
hourlyTT = retime(meanTT,'hourly','linear')
```

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 argument. You can specify a time step of any size as a duration or calendarDuration value.

```
regularTT = retime(meanTT, 'regular', 'linear', 'TimeStep', minutes(30))
```

regularTT=11×3 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	69.576	0.027118	1.9576
09-Jun-2016 07:00:00	66.027	0.044867	1.6027
09-Jun-2016 07:30:00	62.477	0.062616	1.2477
09-Jun-2016 08:00:00	59.158	0.079133	0.9223
09-Jun-2016 08:30:00	66.841	0.03695	2.007
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 into Array

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, as long as the table variables have data types that allow them to be concatenated.

```
A = regularTT.Variables
```

A = 11×3

```
65.8260    0.0522    3.0385
69.3504    0.0315    2.6757
72.8747    0.0107    2.3129
69.5764    0.0271    1.9576
66.0266    0.0449    1.6027
62.4768    0.0626    1.2477
59.1579    0.0791    0.9223
66.8412    0.0370    2.0070
70.2868    0.0133    2.8171
66.2348    0.0226    3.1412
:
```

`regularTT.Variables` is equivalent to using curly brace syntax, `regularTT{:, :}`, to access the data in the timetable variables.

```
A2 = regularTT{:, :}
```

A2 = 11×3

```
65.8260    0.0522    3.0385
69.3504    0.0315    2.6757
```

```
72.8747    0.0107    2.3129
69.5764    0.0271    1.9576
66.0266    0.0449    1.6027
62.4768    0.0626    1.2477
59.1579    0.0791    0.9223
66.8412    0.0370    2.0070
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 10-5
- “Combine Timetables and Synchronize Their Data” on page 10-8
- “Retime and Synchronize Timetable Variables Using Different Methods” on page 10-14
- “Select Times in Timetable” on page 10-19

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];
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=8×3 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
  {'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=8×3 timetable
    Date      X      Y      Z
    _____
    -        -      -      -
```

	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')
```

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'})
```

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'})
```

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',:)
```

		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'})
```

		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
{'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
{'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'})
```

		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`

Loma Prieta Earthquake Analysis

This example shows how to analyze and visualize earthquake data.

Load Earthquake Data

The file `quake.mat` contains 200Hz data from the October 17, 1989 Loma Prieta earthquake in the Santa Cruz Mountains. The data are courtesy of Joel Yellin at the Charles F. Richter Seismological Laboratory, University of California, Santa Cruz.

Start by loading the data.

```
load quake
whos e n v
```

Name	Size	Bytes	Class	Attributes
e	10001x1	80008	double	
n	10001x1	80008	double	
v	10001x1	80008	double	

In the workspace there are three variables, containing time traces from an accelerometer located in the Natural Sciences building at UC Santa Cruz. The accelerometer recorded the main shock amplitude of the earthquake wave. The variables `n`, `e`, `v` refer to the three directional components measured by the instrument, which was aligned parallel to the fault, with its N direction pointing in the direction of Sacramento. The data are uncorrected for the response of the instrument.

Create a variable, `Time`, containing the timestamps sampled at 200Hz with the same length as the other vectors. Represent the correct units with the `seconds` function and multiplication to achieve the Hz (s^{-1}) sampling rate. This results in a `duration` variable which is useful for representing elapsed time.

```
Time = (1/200)*seconds(1:length(e))';
whos Time
```

Name	Size	Bytes	Class	Attributes
Time	10001x1	80010	duration	

Organize Data in Timetable

Separate variables can be organized in a `table` or `timetable` for more convenience. A `timetable` provides flexibility and functionality for working with time-stamped data. Create a `timetable` with the time and three acceleration variables and supply more meaningful variable names. Display the first eight rows using the `head` function.

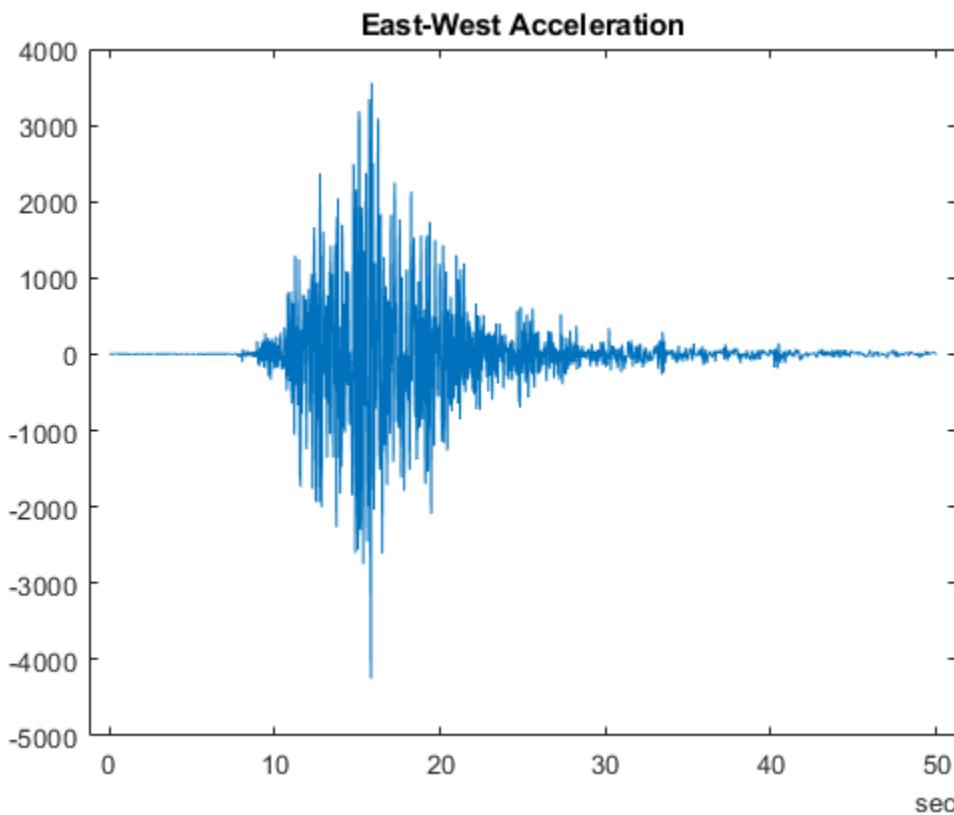
```
varNames = {'EastWest', 'NorthSouth', 'Vertical'};
quakeData = timetable(Time, e, n, v, 'VariableNames', varNames);
head(quakeData)
```

Time	EastWest	NorthSouth	Vertical
0.005 sec	5	3	0
0.01 sec	5	3	0
0.015 sec	5	2	0

0.02 sec	5	2	0
0.025 sec	5	2	0
0.03 sec	5	2	0
0.035 sec	5	1	0
0.04 sec	5	1	0

Explore the data by accessing the variables in the timetable with dot subscripting. (For more information on dot subscripting, see Access Data in a Table.) We chose "East-West" amplitude and plot it as function of the duration.

```
plot(quakeData.Time,quakeData.EastWest)
title('East-West Acceleration')
```



Scale Data

Scale the data by the gravitational acceleration, or multiply each variable in the table by the constant. Since the variables are all of the same type (`double`), you can access all variables using the dimension name, `Variables`. Note that `quakeData.Variables` provides a direct way to modify the numerical values within the timetable.

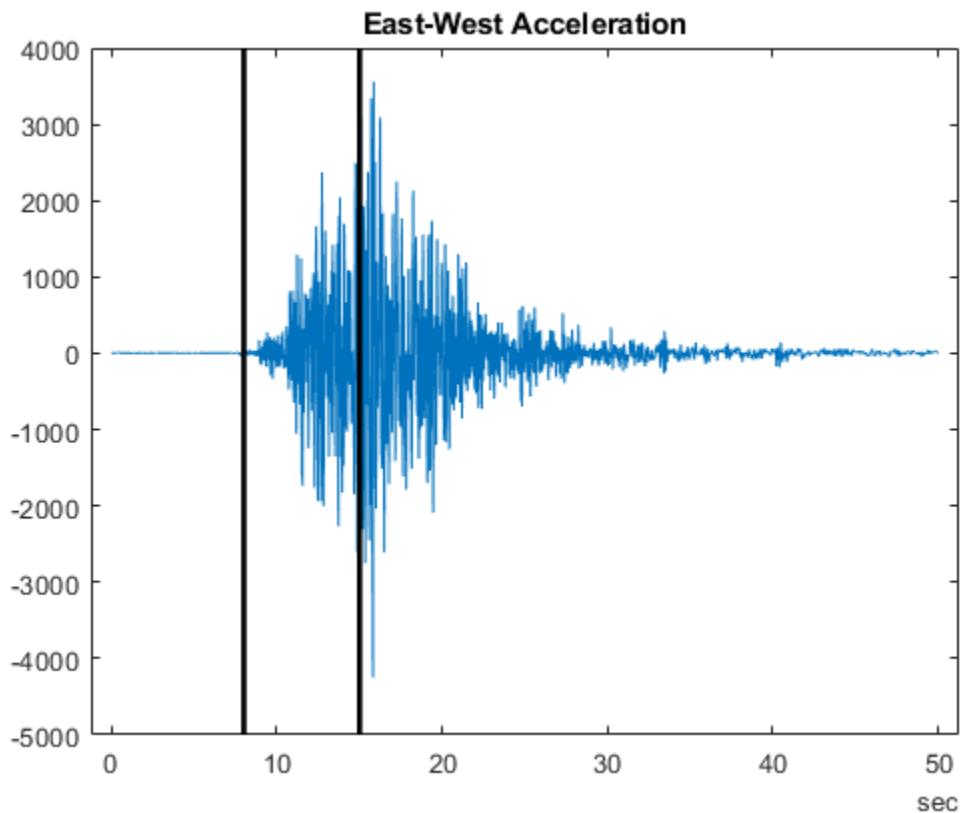
```
quakeData.Variables = 0.098*quakeData.Variables;
```

Select Subset of Data for Exploration

We are interested in the time region where the amplitude of the shockwave starts to increase from near zero to maximum levels. Visual inspection of the above plot shows that the time interval from 8

to 15 seconds is of interest. For better visualization we draw black lines at the selected time spots to draw attention to that interval. All subsequent calculations will involve this interval.

```
t1 = seconds(8)*[1;1];
t2 = seconds(15)*[1;1];
hold on
plot([t1 t2],ylim,'k','LineWidth',2)
hold off
```



Store Data of Interest

Create another timetable with data in this interval. Use `timerange` to select the rows of interest.

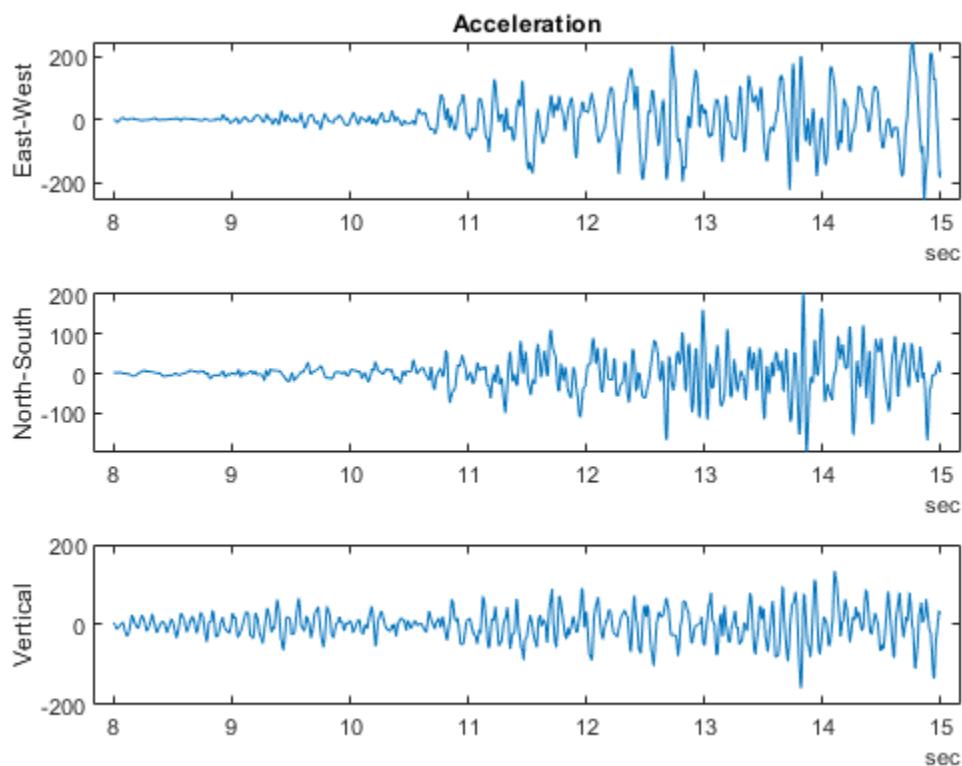
```
tr = timerange(seconds(8),seconds(15));
dataOfInterest = quakeData(tr,:);
head(dataOfInterest)
```

Time	EastWest	NorthSouth	Vertical
8 sec	-0.098	2.254	5.88
8.005 sec	0	2.254	3.332
8.01 sec	-2.058	2.352	-0.392
8.015 sec	-4.018	2.352	-4.116
8.02 sec	-6.076	2.45	-7.742
8.025 sec	-8.036	2.548	-11.466
8.03 sec	-10.094	2.548	-9.8

8.035 sec -8.232 2.646 -8.134

Visualize the three acceleration variables on three separate axes.

```
figure
subplot(3,1,1)
plot(dataOfInterest.Time,dataOfInterest.EastWest)
ylabel('East-West')
title('Acceleration')
subplot(3,1,2)
plot(dataOfInterest.Time,dataOfInterest.NorthSouth)
ylabel('North-South')
subplot(3,1,3)
plot(dataOfInterest.Time,dataOfInterest.Vertical)
ylabel('Vertical')
```



Calculate Summary Statistics

To display statistical information about the data we use the `summary` function.

```
summary(dataOfInterest)
```

RowTimes:

```
Time: 1400x1 duration
Values:
    Min           8 sec
```

```

Median      11.498 sec
Max        14.995 sec
TimeStep    0.005 sec

```

Variables:

EastWest: 1400x1 double

Values:

```

Min      -255.09
Median   -0.098
Max       244.51

```

NorthSouth: 1400x1 double

Values:

```

Min      -198.55
Median   1.078
Max       204.33

```

Vertical: 1400x1 double

Values:

```

Min      -157.88
Median   0.98
Max       134.46

```

Additional statistical information about the data can be calculated using `varfun`. This is useful for applying functions to each variable in a table or timetable. The function to apply is passed to `varfun` as a function handle. Below we apply the `mean` function to all three variables and output the result in format of a table, since the time is not meaningful after computing the temporal means.

```

mn = varfun(@mean,dataOfInterest,'OutputFormat','table')

mn=1x3 table
  mean_EastWest     mean_NorthSouth     mean_Vertical
  _____
  0.9338           -0.10276          -0.52542

```

Calculate Velocity and Position

To identify the speed of propagation of the shockwave, we integrate the accelerations once. We use cumulative sums along the time variable to get the velocity of the wave front.

```

edot = (1/200)*cumsum(dataOfInterest.EastWest);
edot = edot - mean(edot);

```

Below we perform the integration on all three variables to calculate the velocity. It is convenient to create a function and apply it to the variables in the `timetable` with `varfun`. In this example, we included the function at the end of this file and named it `velFun`.

```

vel = varfun(@velFun,dataOfInterest);
head(vel)

```

Time	velFun_EastWest	velFun_NorthSouth	velFun_Vertical
8 sec	-0.56831	0.44642	1.8173
8.005 sec	-0.56831	0.45769	1.834
8.01 sec	-0.5786	0.46945	1.832
8.015 sec	-0.59869	0.48121	1.8114
8.02 sec	-0.62907	0.49346	1.7727
8.025 sec	-0.66925	0.5062	1.7154
8.03 sec	-0.71972	0.51894	1.6664
8.035 sec	-0.76088	0.53217	1.6257

Apply the same function `velFun` to the velocities to determine the position.

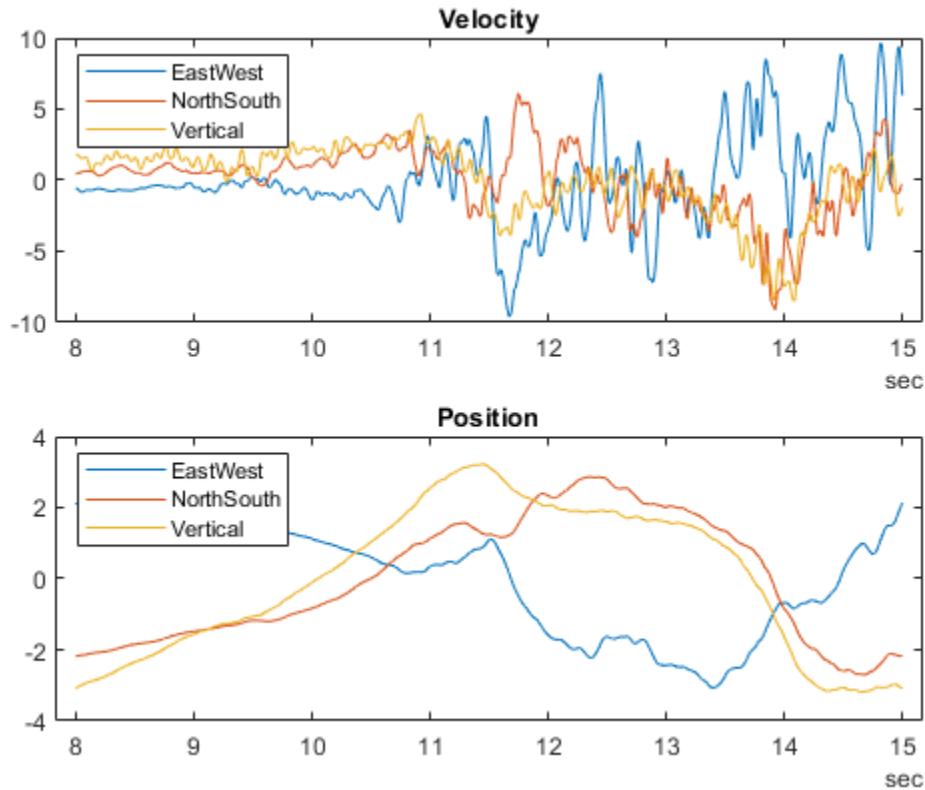
Time	velFun_velFun_EastWest	velFun_velFun_NorthSouth	velFun_velFun_Vertical
8 sec	2.1189	-2.1793	-3.0821
8.005 sec	2.1161	-2.177	-3.0729
8.01 sec	2.1132	-2.1746	-3.0638
8.015 sec	2.1102	-2.1722	-3.0547
8.02 sec	2.107	-2.1698	-3.0458
8.025 sec	2.1037	-2.1672	-3.0373
8.03 sec	2.1001	-2.1646	-3.0289
8.035 sec	2.0963	-2.162	-3.0208

Notice how the variable names in the timetable created by `varfun` include the name of the function used. It is useful to track the operations that have been performed on the original data. Adjust the variable names back to their original values using dot notation.

```
pos.Properties.VariableNames = varNames;
```

Below we plot the 3 components of the velocity and position for the time interval of interest.

```
figure
subplot(2,1,1)
plot(vel.Time,vel.Variables)
legend(quakeData.Properties.VariableNames,'Location','NorthWest')
title('Velocity')
subplot(2,1,2)
plot(vel.Time,pos.Variables)
legend(quakeData.Properties.VariableNames,'Location','NorthWest')
title('Position')
```

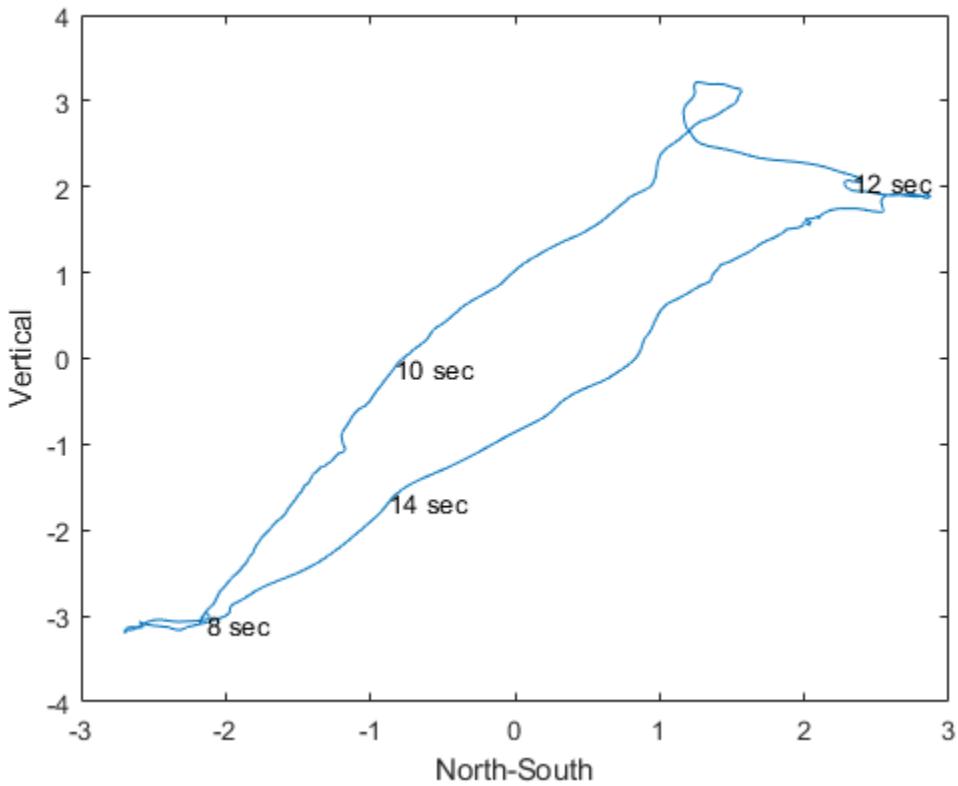


Visualize Trajectories

The trajectories can be plotted in 2D or 3D by using the component value. In the following we will show different ways of visualizing this data.

Begin with 2-dimensional projections. Here is the first with a few values of time annotated.

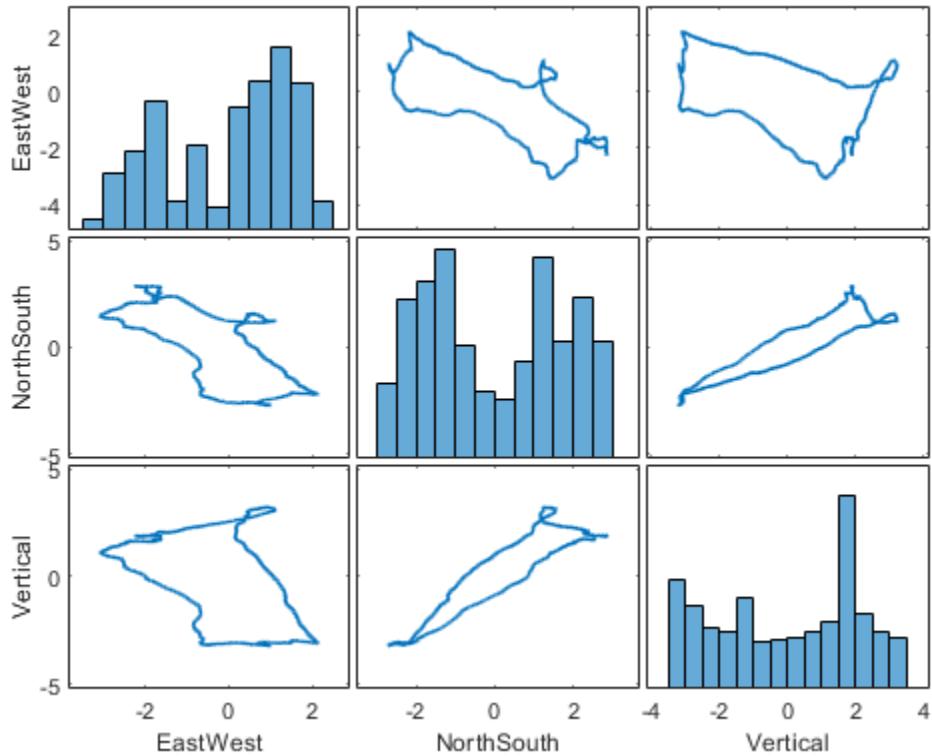
```
figure
plot(pos.NorthSouth,pos.Vertical)
xlabel('North-South')
ylabel('Vertical')
% Select locations and label
nt = ceil((max(pos.Time) - min(pos.Time))/6);
idx = find(fix(pos.Time/nt) == (pos.Time/nt))';
text(pos.NorthSouth(idx),pos.Vertical(idx),char(pos.Time(idx)))
```



Use `plotmatrix` to visualize a grid of scatter plots of all variables against one another and histograms of each variable on the diagonal. The output variable `Ax`, represents each axes in the grid and can be used to identify which axes to label using `xlabel` and `ylabel`.

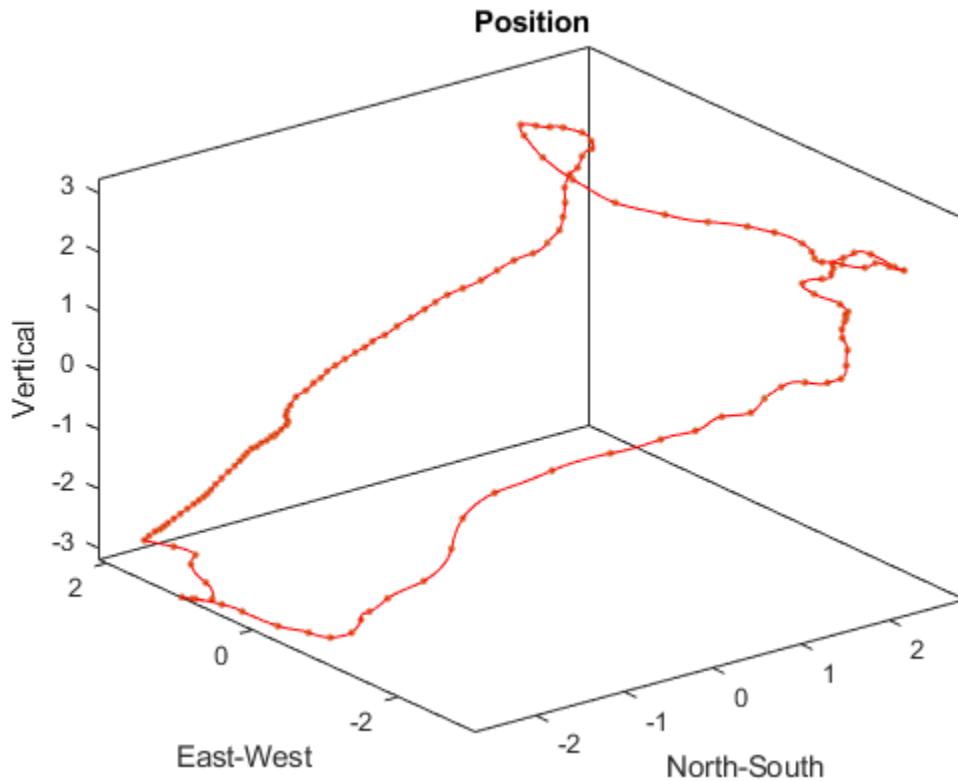
```
figure
[S,Ax] = plotmatrix(pos.Variables);

for ii = 1:length(varNames)
    xlabel(Ax(end,ii),varNames{ii})
    ylabel(Ax(ii,1),varNames{ii})
end
```



Plot a 3-D view of the trajectory and plot a dot at every tenth position point. The spacing between dots indicates the velocity.

```
step = 10;
figure
plot3(pos.NorthSouth,pos.EastWest,pos.Vertical, 'r')
hold on
plot3(pos.NorthSouth(1:step:end),pos.EastWest(1:step:end),pos.Vertical(1:step:end), 'r.')
hold off
box on
axis tight
xlabel('North-South')
ylabel('East-West')
zlabel('Vertical')
title('Position')
```



Supporting Functions

Functions are defined below.

```
function y = velFun(x)
    y = (1/200)*cumsum(x);
    y = y - mean(y);
end
```

See Also

[duration](#) | [head](#) | [seconds](#) | [summary](#) | [timerange](#) | [timetable](#) | [varfun](#)

Related Examples

- “Represent Dates and Times in MATLAB” on page 7-2
- “Create Timetables” on page 10-2
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27
- “Select Times in Timetable” on page 10-19
- “Access Data in Tables” on page 9-34

Preprocess and Explore Time-stamped Data Using timetable

This example shows how to analyze bicycle traffic patterns from sensor data using the `timetable` data container to organize and preprocess time-stamped data. The data come from sensors on Broadway Street in Cambridge, MA. The City of Cambridge provides public access to the full data set at the Cambridge Open Data site.

This example shows how to perform a variety of data cleaning, munging, and preprocessing tasks such as removing missing values and synchronizing time-stamped data with different timesteps. In addition, data exploration is highlighted including visualizations and grouped calculations using the `timetable` data container to:

- Explore daily bicycle traffic
- Compare bicycle traffic to local weather conditions
- Analyze bicycle traffic on various days of the week and times of day

Import Bicycle Traffic Data into Timetable

Import a sample of the bicycle traffic data from a comma-separated text file. The `readtable` function returns the data in a table. Display the first eight rows using the `head` function.

```
bikeTbl = readtable('BicycleCounts.csv');
head(bikeTbl)
```

Timestamp	Day	Total	Westbound	Eastbound
2015-06-24 00:00:00	{'Wednesday'}	13	9	4
2015-06-24 01:00:00	{'Wednesday'}	3	3	0
2015-06-24 02:00:00	{'Wednesday'}	1	1	0
2015-06-24 03:00:00	{'Wednesday'}	1	1	0
2015-06-24 04:00:00	{'Wednesday'}	1	1	0
2015-06-24 05:00:00	{'Wednesday'}	7	3	4
2015-06-24 06:00:00	{'Wednesday'}	36	6	30
2015-06-24 07:00:00	{'Wednesday'}	141	13	128

The data have timestamps, so it is convenient to use a timetable to store and analyze the data. A timetable is similar to a table, but includes timestamps that are associated with the rows of data. The timestamps, or row times, are represented by `datetime` or `duration` values. `datetime` and `duration` are the recommended data types for representing points in time or elapsed times, respectively.

Convert `bikeTbl` into a timetable using the `table2timetable` function. You must use a conversion function because `readtable` returns a table. `table2timetable` converts the first `datetime` or `duration` variable in the table into the row times of a timetable. The row times are metadata that label the rows. However, when you display a timetable, the row times and timetable variables are displayed in a similar fashion. Note that the table has five variables whereas the timetable has four.

```
bikeData = table2timetable(bikeTbl);
head(bikeData)
```

Timestamp	Day	Total	Westbound	Eastbound
2015-06-24 00:00:00	{'Wednesday'}	13	9	4

2015-06-24 00:00:00	{'Wednesday'}	13	9	4
2015-06-24 01:00:00	{'Wednesday'}	3	3	0
2015-06-24 02:00:00	{'Wednesday'}	1	1	0
2015-06-24 03:00:00	{'Wednesday'}	1	1	0
2015-06-24 04:00:00	{'Wednesday'}	1	1	0
2015-06-24 05:00:00	{'Wednesday'}	7	3	4
2015-06-24 06:00:00	{'Wednesday'}	36	6	30
2015-06-24 07:00:00	{'Wednesday'}	141	13	128

```
whos bikeTbl bikeData
```

Name	Size	Bytes	Class	Attributes
bikeData	9387x4	1412425	timetable	
bikeTbl	9387x5	1487735	table	

Access Times and Data

Convert the Day variable to categorical. The categorical data type is designed for data that consists of a finite set of discrete values, such as the names of the days of the week. List the categories so they display in day order. Use dot subscripting to access variables by name.

```
bikeData.Day = categorical(bikeData.Day, {'Sunday', 'Monday', 'Tuesday', ...
    'Wednesday', 'Thursday', 'Friday', 'Saturday'});
```

In a timetable, the times are treated separately from the data variables. Access the **Properties** of the timetable to show that the row times are the first dimension of the timetable, and the variables are the second dimension. The **DimensionNames** property shows the names of the two dimensions, while the **VariableNames** property shows the names of the variables along the second dimension.

```
bikeData.Properties
```

```
ans =
TimetableProperties with properties:

    Description: ''
    UserData: []
    DimensionNames: {'Timestamp'  'Variables'}
    VariableNames: {'Day'  'Total'  'Westbound'  'Eastbound'}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: []
        RowTimes: [9387x1 datetime]
        StartTime: 2015-06-24 00:00:00
        SampleRate: NaN
        TimeStep: NaN
    CustomProperties: No custom properties are set.
    Use addprop and rmprop to modify CustomProperties.
```

By default, **table2timetable** assigned **Timestamp** as the first dimension name when it converted the table to a timetable, since this was the variable name from the original table. You can change the names of the dimensions, and other timetable metadata, through the **Properties**.

Change the names of the dimensions to **Time** and **Data**.

```

bikeData.Properties.DimensionNames = {'Time' 'Data'};
bikeData.Properties

ans =
TimetableProperties with properties:

    Description: ''
    UserData: []
    DimensionNames: {'Time' 'Data'}
    VariableNames: {'Day' 'Total' 'Westbound' 'Eastbound'}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: []
    RowTimes: [9387x1 datetime]
    StartTime: 2015-06-24 00:00:00
    SampleRate: NaN
    TimeStep: NaN
    CustomProperties: No custom properties are set.
    Use addprop and rmprop to modify CustomProperties.

```

Display the first eight rows of the timetable.

```
head(bikeData)
```

Time	Day	Total	Westbound	Eastbound
2015-06-24 00:00:00	Wednesday	13	9	4
2015-06-24 01:00:00	Wednesday	3	3	0
2015-06-24 02:00:00	Wednesday	1	1	0
2015-06-24 03:00:00	Wednesday	1	1	0
2015-06-24 04:00:00	Wednesday	1	1	0
2015-06-24 05:00:00	Wednesday	7	3	4
2015-06-24 06:00:00	Wednesday	36	6	30
2015-06-24 07:00:00	Wednesday	141	13	128

Determine the number of days that elapsed between the latest and earliest row times. The variables can be accessed by dot notation when referencing variables one at a time.

```
elapsedTime = max(bikeData.Time) - min(bikeData.Time)
```

```
elapsedTime = duration
9383:30:00
```

```
elapsedTime.Format = 'd'
```

```
elapsedTime = duration
390.98 days
```

To examine the typical bicycle counts on a given day, calculate the means for the total number of bikes, and the numbers travelling westbound and eastbound.

Return the numeric data as a matrix by indexing into the contents of `bikeData` using curly braces. Display the first eight rows. Use standard table subscripting to access multiple variables.

```
counts = bikeData{:,2:end};  
counts(1:8,:)
```

```
ans = 8x3
```

```
13    9    4  
 3    3    0  
 1    1    0  
 1    1    0  
 1    1    0  
 7    3    4  
36    6   30  
141   13  128
```

Since the mean is appropriate for only the numeric data, you can use the `vartype` function to select the numeric variables. `vartype` can be more convenient than manually indexing into a table or timetable to select variables. Calculate the means and omit `Nan` values.

```
counts = bikeData{:,vartype('numeric')};  
mean(counts,'omitnan')
```

```
ans = 1x3
```

```
49.8860  24.2002  25.6857
```

Select Data by Date and Time of Day

To determine how many people bicycle during a holiday, examine the data on the 4th of July holiday. Index into the timetable by row times for July 4, 2015. When you index on row times, you must match times exactly. You can specify the time indices as `datetime` or `duration` values, or as character vectors that can be converted to dates and times. You can specify multiple times as an array.

Index into `bikeData` with specific dates and times to extract data for July 4, 2015. If you specify the date only, then the time is assumed to be midnight, or 00:00:00.

```
bikeData('2015-07-04',:)
```

```
ans=1x4 timetable  
Time          Day       Total      Westbound     Eastbound  
_____|_____|_____|_____|_____|  
2015-07-04 00:00:00 Saturday    8           7           1
```

```
d = {'2015-07-04 08:00:00','2015-07-04 09:00:00'};  
bikeData(d,:)
```

```
ans=2x4 timetable  
Time          Day       Total      Westbound     Eastbound  
_____|_____|_____|_____|_____|  
2015-07-04 08:00:00 Saturday   15           3           12  
2015-07-04 09:00:00 Saturday   21           4           17
```

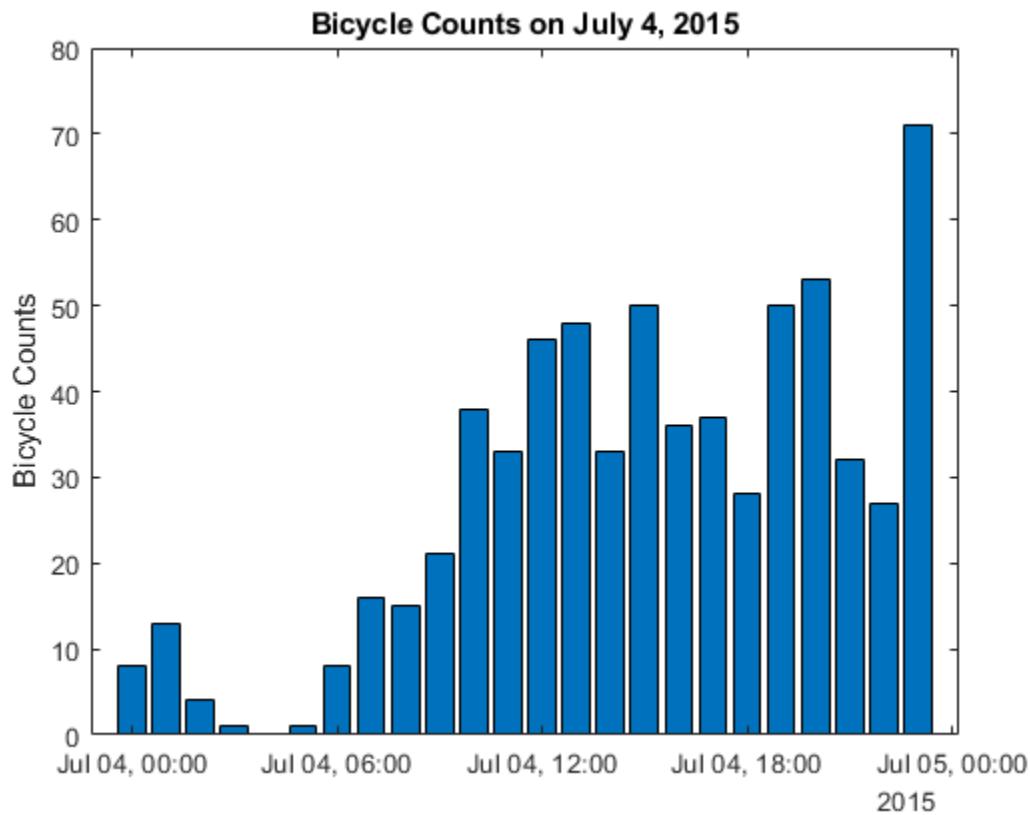
It would be tedious to use this strategy to extract the entire day. You can also specify ranges of time without indexing on specific times. To create a time range subscript as a helper, use the `timerange` function.

Subscript into the timetable using a time range for the entire day of July 4, 2015. Specify the start time as midnight on July 4, and the end time as midnight on July 5. By default, `timerange` covers all times starting with the start time and up to, but not including, the end time. Plot the bicycle counts over the course of the day.

```
tr = timerange('2015-07-04','2015-07-05');
jul4 = bikeData(tr,'Total');
head(jul4)
```

```
ans=8×1 timetable
    Time          Total
    _____
    2015-07-04 00:00:00      8
    2015-07-04 01:00:00     13
    2015-07-04 02:00:00      4
    2015-07-04 03:00:00      1
    2015-07-04 04:00:00      0
    2015-07-04 05:00:00      1
    2015-07-04 06:00:00      8
    2015-07-04 07:00:00     16
```

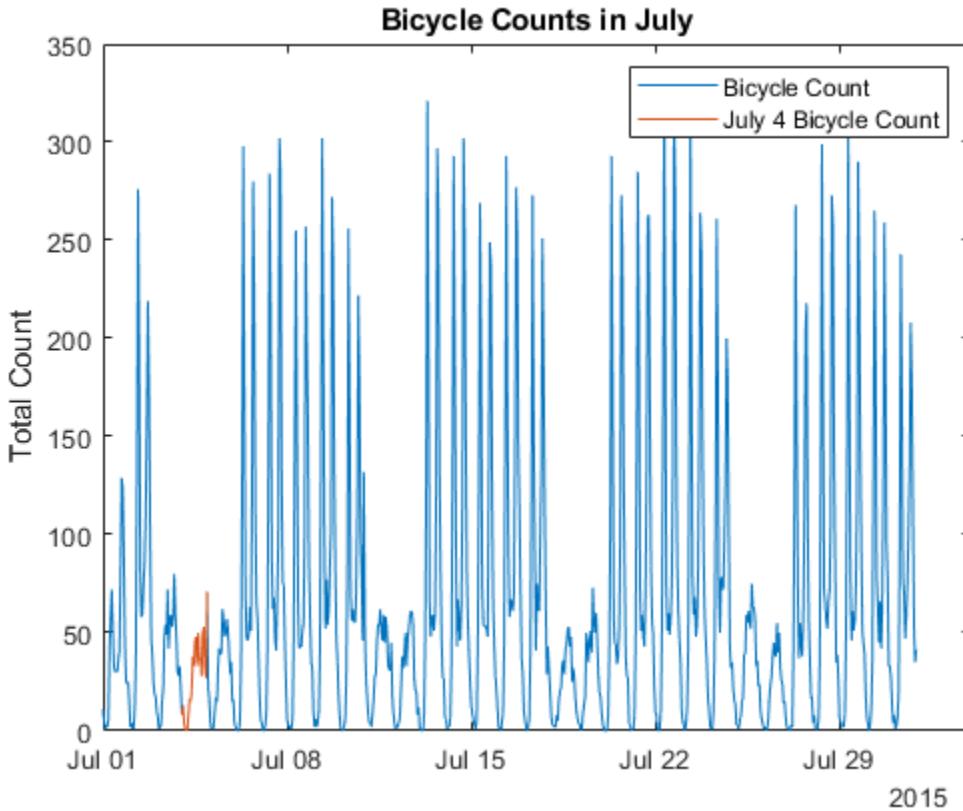
```
bar(jul4.Time,jul4.Total)
ylabel('Bicycle Counts')
title('Bicycle Counts on July 4, 2015')
```



From the plot, there is more volume throughout the day, leveling off in the afternoon. Because many businesses are closed, the plot does not show typical traffic during commute hours. Spikes later in the evening can be attributed to celebrations with fireworks, which occur after dark. To examine these trends more closely, the data should be compared to data for typical days.

Compare the data for July 4 to data for the rest of the month of July.

```
jul = bikeData(timerange('2015-07-01','2015-08-01'),:);
plot(jul.Time,jul.Total)
hold on
plot(jul4.Time,jul4.Total)
ylabel('Total Count')
title('Bicycle Counts in July')
hold off
legend('Bicycle Count','July 4 Bicycle Count')
```



The plot shows variations that can be attributed to traffic differences between weekdays and weekends. The traffic patterns for July 4 and 5 are consistent with the pattern for weekend traffic. July 5 is a Monday but is often observed as a holiday. These trends can be examined more closely with further preprocessing and analysis.

Preprocess Times and Data Using timetable

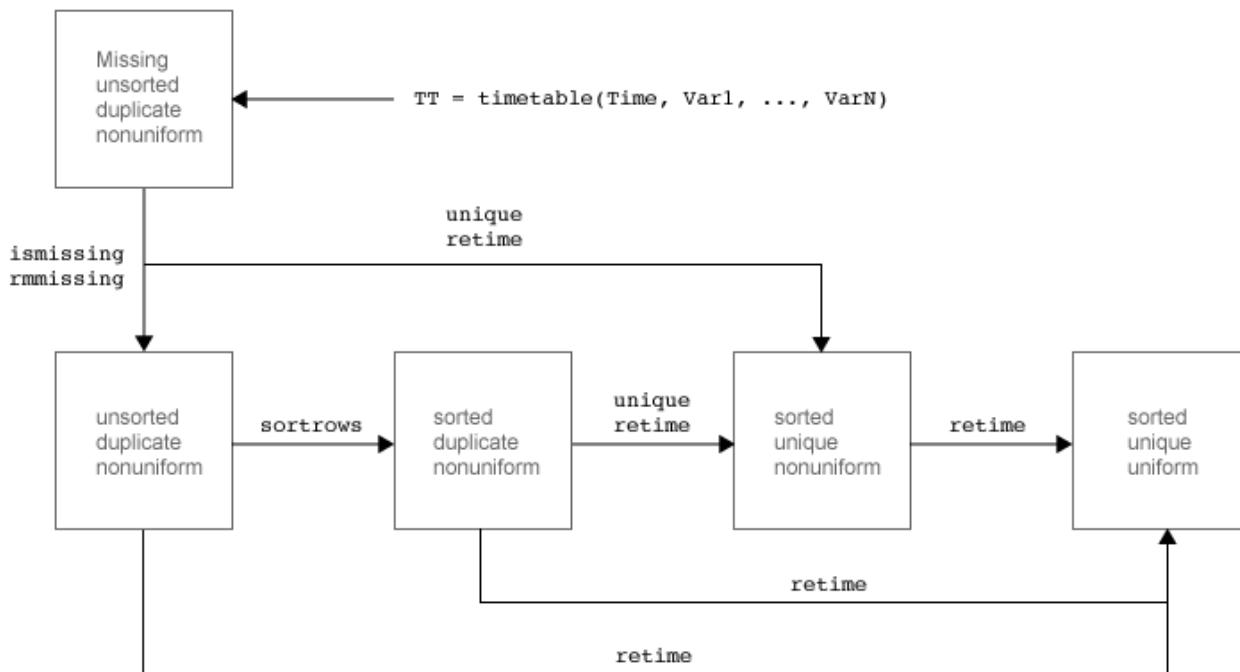
Time-stamped data sets are often messy and may contain anomalies or errors. Timetables are well suited for resolving anomalies and errors.

A timetable does not have to have its row times in any particular order. It can contain rows that are not sorted by their row times. A timetable can also 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. A timetable can even contain `NaT` or `NaN` values to indicate missing row times.

The `timetable` data type provides a number of different ways to resolve missing, duplicate, or nonuniform times. You can also resample or aggregate data to create a *regular* timetable. When a timetable is regular, it has row times that are sorted and unique, and have a uniform or evenly spaced time step between them.

- 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 make a regular timetable, specify a uniformly spaced time vector and use **`retime`**.



Sort in Time Order

Determine if the timetable is sorted. A timetable is sorted if its row times are listed in ascending order.

```
issorted(bikeData)  
ans = logical  
    0
```

Sort the timetable. The `sortrows` function sorts the rows by their row times, from earliest to latest time. If there are rows with duplicate row times, then `sortrows` copies all the duplicates to the output.

```
bikeData = sortrows(bikeData);  
issorted(bikeData)  
  
ans = logical  
    1
```

Identify and Remove Missing Times and Data

A timetable can have missing data indicators in its variables or its row times. For example, you can indicate missing numeric values as NaNs, and missing datetime values as NaTs. You can assign, find,

remove, and fill missing values with the `standardizeMissing`, `ismissing`, `rmmissing`, and `fillmissing` functions, respectively.

Find and count the missing values in the timetable variables. In this example, missing values indicate circumstances when no data were collected.

```
missData = ismissing(bikeData);
sum(missData)
```

```
ans = 1x4
```

```
1     3     3     3
```

The output from `ismissing` is a logical matrix, the same size as the table, identifying missing data values as true. Display any rows which have missing data indicators.

```
idx = any(missData,2);
bikeData(idx,:)
```

ans=3x4 timetable	Time	Day	Total	Westbound	Eastbound
2015-08-03 00:00:00	Monday	NaN	NaN	NaN	
2015-08-03 01:00:00	Monday	NaN	NaN	NaN	
NaT	<undefined>	NaN	NaN	NaN	

`ismissing(bikeData)` finds missing data in the timetable variables only, not the times. To find missing row times, call `ismissing` on the row times.

```
missTimes = ismissing(bikeData.Time);
bikeData(missTimes,:)
```

ans=2x4 timetable	Time	Day	Total	Westbound	Eastbound
NaT	<undefined>	NaN	NaN	NaN	
NaT	Friday	6	3	3	

In this example, missing times or data values indicate measurement errors and can be excluded. Remove rows of the table containing missing data values and missing row times using `rmmissing`.

```
bikeData = rmmissing(bikeData);
sum(ismissing(bikeData))
```

```
ans = 1x4
```

```
0     0     0     0
```

```
sum(ismissing(bikeData.Time))
```

```
ans = 0
```

Remove Duplicate Times and Data

Determine if there are duplicate times and/or duplicate rows of data. You might want to exclude exact duplicates, as these can also be considered measurement errors. Identify duplicate times by finding where the difference between the sorted times is exactly zero.

```
idx = diff(bikeData.Time) == 0;
dup = bikeData.Time(idx)
```

```
dup = 3x1 datetime
2015-08-21 00:00:00
2015-11-19 23:00:00
2015-11-19 23:00:00
```

Three times are repeated and November 19, 2015 is repeated twice. Examine the data associated with the repeated times.

```
bikeData(dup(1),:)
```

```
ans=2x4 timetable
Time          Day       Total   Westbound  Eastbound
-----|-----|-----|-----|-----|
2015-08-21 00:00:00 | Friday    14      9        5
2015-08-21 00:00:00 | Friday    11      7        4
```

```
bikeData(dup(2),:)
```

```
ans=3x4 timetable
Time          Day       Total   Westbound  Eastbound
-----|-----|-----|-----|-----|
2015-11-19 23:00:00 | Thursday  17      15        2
2015-11-19 23:00:00 | Thursday  17      15        2
2015-11-19 23:00:00 | Thursday  17      15        2
```

The first has duplicated times but non-duplicate data, whereas the others are entirely duplicated. Timetable rows are considered duplicates when they contain identical row times and identical data values across the rows. You can use `unique` to remove duplicate rows in the timetable. The `unique` function also sorts the rows by their row times.

```
bikeData = unique(bikeData);
```

The rows with duplicate times but non-duplicate data require some interpretation. Examine the data around those times.

```
d = dup(1) + hours(-2:2);
bikeData(d,:)
```

```
ans=5x4 timetable
Time          Day       Total   Westbound  Eastbound
-----|-----|-----|-----|-----|
2015-08-20 22:00:00 | Thursday  40      30        10
2015-08-20 23:00:00 | Thursday  25      18         7
```

2015-08-21 00:00:00	Friday	11	7	4
2015-08-21 00:00:00	Friday	14	9	5
2015-08-21 02:00:00	Friday	6	5	1

In this case, the duplicate time may have been mistaken since the data and surrounding times are consistent. Though it appears to represent 01:00:00, it is uncertain what time this should have been. The data can be accumulated to account for the data at both time points.

```
sum(bikeData{dup(1),2:end})
```

```
ans = 1x3
```

```
25     16      9
```

This is only one case which can be done manually. However, for many rows, the `retime` function can perform this calculation. Accumulate the data for the unique times using the `sum` function to aggregate. The sum is appropriate for numeric data but not the categorical data in the timetable. Use `vartype` to identify the numeric variables.

```
vt = vartype('numeric');
t = unique(bikeData.Time);
numData = retime(bikeData(:,vt),t,'sum');
head(numData)
```

```
ans=8x3 timetable
```

Time	Total	Westbound	Eastbound
2015-06-24 00:00:00	13	9	4
2015-06-24 01:00:00	3	3	0
2015-06-24 02:00:00	1	1	0
2015-06-24 03:00:00	1	1	0
2015-06-24 04:00:00	1	1	0
2015-06-24 05:00:00	7	3	4
2015-06-24 06:00:00	36	6	30
2015-06-24 07:00:00	141	13	128

You cannot sum the categorical data, but since one label represents the whole day, take the first value on each day. You can perform the `retime` operation again with the same time vector and concatenate the timetables together.

```
vc = vartype('categorical');
catData = retime(bikeData(:,vc),t,'firstvalue');
bikeData = [catData,numData];
bikeData(d,:)
```

```
ans=4x4 timetable
```

Time	Day	Total	Westbound	Eastbound
2015-08-20 22:00:00	Thursday	40	30	10
2015-08-20 23:00:00	Thursday	25	18	7
2015-08-21 00:00:00	Friday	25	16	9
2015-08-21 02:00:00	Friday	6	5	1

Examine Uniformity of Time Interval

The data appear to have a uniform time step of one hour. To determine if this is true for all the row times in the timetable, use the `isregular` function. `isregular` returns `true` for sorted, evenly-spaced times (monotonically increasing), with no duplicate or missing times (`NaT` or `Nan`).

```
isregular(bikeData)  
ans = logical  
    0
```

The output of `0`, or `false`, indicates that the times in the timetable are not evenly spaced. Explore the time interval in more detail.

```
dt = diff(bikeData.Time);  
[min(dt); max(dt)]  
  
ans = 2x1 duration  
00:30:00  
03:00:00
```

To put the timetable on a regular time interval, use `retime` or `synchronize` and specify the time interval of interest.

Determine Daily Bicycle Volume

Determine the counts per day using the `retime` function. Accumulate the count data for each day using the `sum` method. This is appropriate for numeric data but not the categorical data in the timetable. Use `vartype` to identify the variables by data type.

```
dayCountNum = retime(bikeData(:,vt),'daily','sum');  
head(dayCountNum)  
  
ans=8x3 timetable  
Time Total Westbound Eastbound  
_____ _____ _____  
2015-06-24 00:00:00 2141 1141 1000  
2015-06-25 00:00:00 2106 1123 983  
2015-06-26 00:00:00 1748 970 778  
2015-06-27 00:00:00 695 346 349  
2015-06-28 00:00:00 153 83 70  
2015-06-29 00:00:00 1841 978 863  
2015-06-30 00:00:00 2170 1145 1025  
2015-07-01 00:00:00 997 544 453
```

As above, you can perform the `retime` operation again to represent the categorical data using an appropriate method and concatenate the timetables together.

```
dayCountCat = retime(bikeData(:,vc),'daily','firstvalue');  
dayCount = [dayCountCat,dayCountNum];  
head(dayCount)  
  
ans=8x4 timetable  
Time Day Total Westbound Eastbound
```

2015-06-24 00:00:00	Wednesday	2141	1141	1000
2015-06-25 00:00:00	Thursday	2106	1123	983
2015-06-26 00:00:00	Friday	1748	970	778
2015-06-27 00:00:00	Saturday	695	346	349
2015-06-28 00:00:00	Sunday	153	83	70
2015-06-29 00:00:00	Monday	1841	978	863
2015-06-30 00:00:00	Tuesday	2170	1145	1025
2015-07-01 00:00:00	Wednesday	997	544	453

Synchronize Bicycle Count and Weather Data

Examine the effect of weather on cycling behavior by comparing the bicycle count with weather data. Load the weather timetable which includes historical weather data from Boston, MA including storm events.

```
load BostonWeatherData
head(weatherData)
```

Time	TemperatureF	Humidity	Events
01-Jul-2015	72	78	Thunderstorm
02-Jul-2015	72	60	None
03-Jul-2015	70	56	None
04-Jul-2015	67	75	None
05-Jul-2015	72	67	None
06-Jul-2015	74	69	None
07-Jul-2015	75	77	Rain
08-Jul-2015	79	68	Rain

To summarize the times and variables in the timetable, use the `summary` function.

```
summary(weatherData)
```

RowTimes:

```
Time: 383x1 datetime
Values:
    Min            01-Jul-2015
    Median        08-Jan-2016
    Max           17-Jul-2016
    TimeStep      24:00:00
```

Variables:

TemperatureF: 383x1 double

Values:

Min	2
Median	55
Max	85

Humidity: 383x1 double

Values:

Min	29
Median	64
Max	97

Events: 383x1 categorical

Values:

Fog	7
Hail	1
Rain	108
Rain-Snow	4
Snow	18
Thunderstorm	12
None	233

Combine the bicycle data with the weather data to a common time vector using `synchronize`. You can resample or aggregate timetable data using any of the methods documented on the reference page for the `synchronize` function.

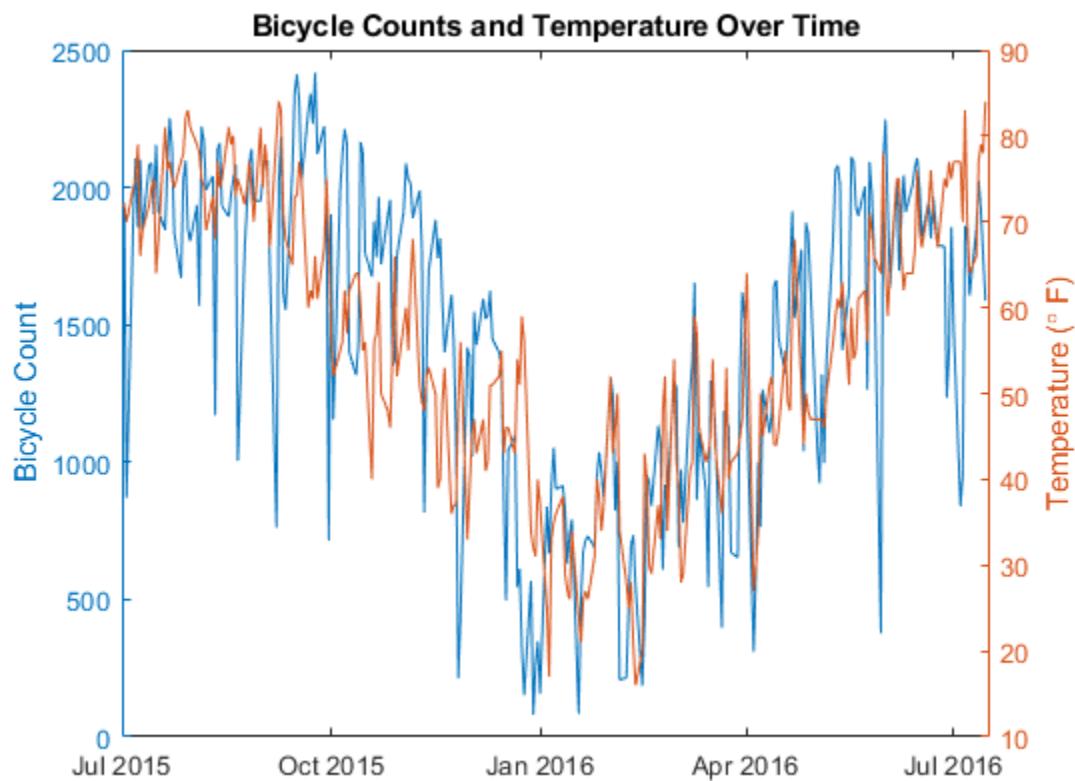
Synchronize the data from both timetables to a common time vector, constructed from the intersection of their individual daily time vectors.

```
data = synchronize(dayCount,weatherData,'intersection');
head(data)
```

Time	Day	Total	Westbound	Eastbound	TemperatureF	Humid
2015-07-01 00:00:00	Wednesday	997	544	453	72	78
2015-07-02 00:00:00	Thursday	1943	1033	910	72	60
2015-07-03 00:00:00	Friday	870	454	416	70	56
2015-07-04 00:00:00	Saturday	669	328	341	67	75
2015-07-05 00:00:00	Sunday	702	407	295	72	67
2015-07-06 00:00:00	Monday	1900	1029	871	74	69
2015-07-07 00:00:00	Tuesday	2106	1140	966	75	77
2015-07-08 00:00:00	Wednesday	1855	984	871	79	68

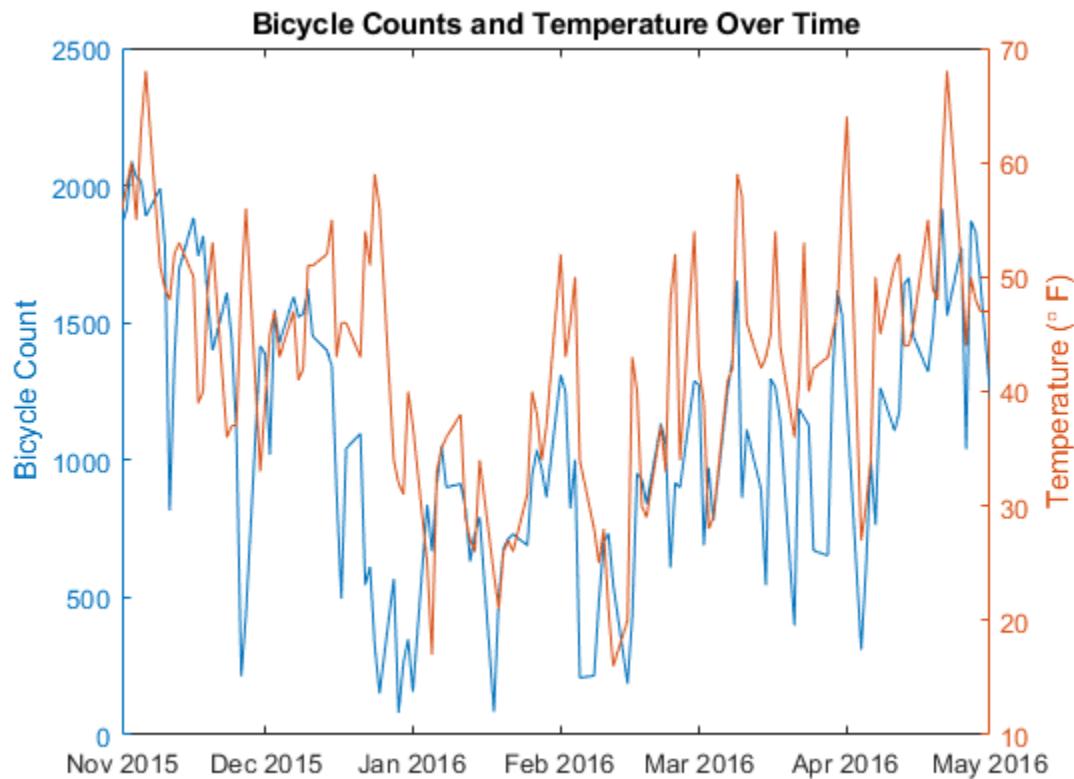
Compare bicycle traffic counts and outdoor temperature on separate y axes to examine the trends. Remove the weekends from the data for visualization.

```
idx = ~isweekend(data.Time);
weekdayData = data(idx,['TemperatureF','Total']);
figure
yyaxis left
plot(weekdayData.Time, weekdayData.Total)
ylabel('Bicycle Count')
yyaxis right
plot(weekdayData.Time,weekdayData.TemperatureF)
ylabel('Temperature (\circ F)')
title('Bicycle Counts and Temperature Over Time')
xlim([min(data.Time) max(data.Time)])
```



The plot shows that the traffic and weather data might follow similar trends. Zoom in on the plot.

```
xlim([datetime('2015-11-01'), datetime('2016-05-01')])
```



The trends are similar, indicating that fewer people cycle on colder days.

Analyze by Day of Week and Time of Day

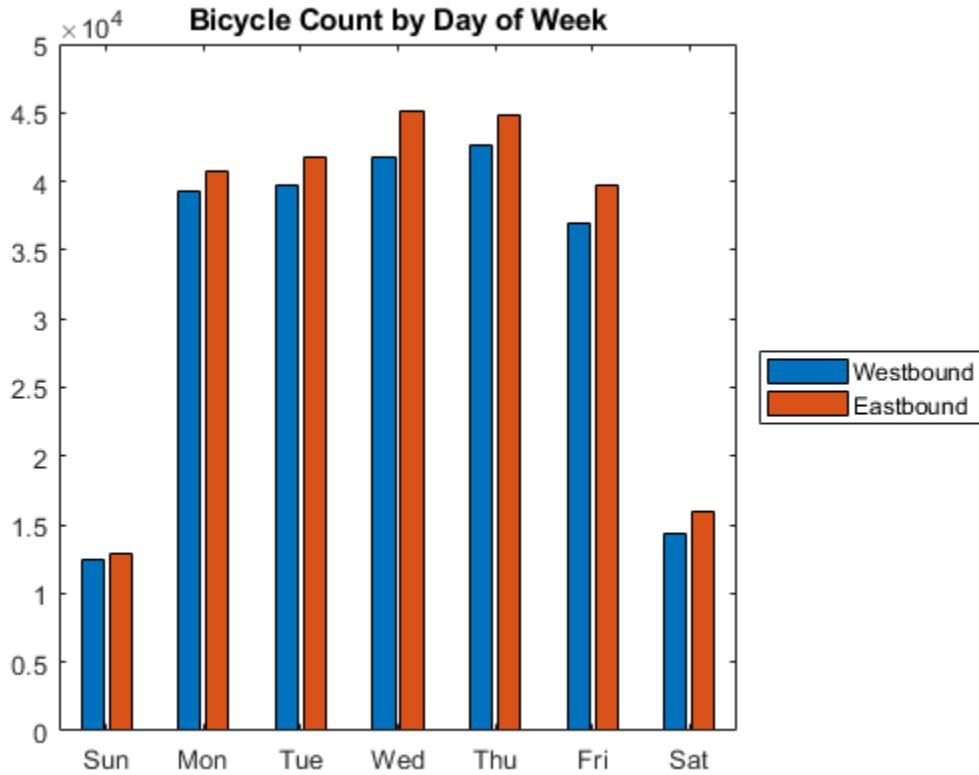
Examine the data based on different intervals such as day of the week and time of day. Determine the total counts per day using `varfun` to perform grouped calculations on variables. Specify the `sum` function with a function handle and the grouping variable and preferred output type using name-value pairs.

```
byDay = varfun(@sum,bikeData, 'GroupingVariables', 'Day', ...
    'OutputFormat', 'table')
```

Day	GroupCount	sum_Total	sum_Westbound	sum_Eastbound
Sunday	1344	25315	12471	12844
Monday	1343	79991	39219	40772
Tuesday	1320	81480	39695	41785
Wednesday	1344	86853	41726	45127
Thursday	1344	87516	42682	44834
Friday	1342	76643	36926	39717
Saturday	1343	30292	14343	15949

```
figure
bar(byDay{:, {'sum_Westbound', 'sum_Eastbound'}})
legend({'Westbound', 'Eastbound'}, 'Location', 'eastoutside')
```

```
xticklabels({'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'})
title('Bicycle Count by Day of Week')
```



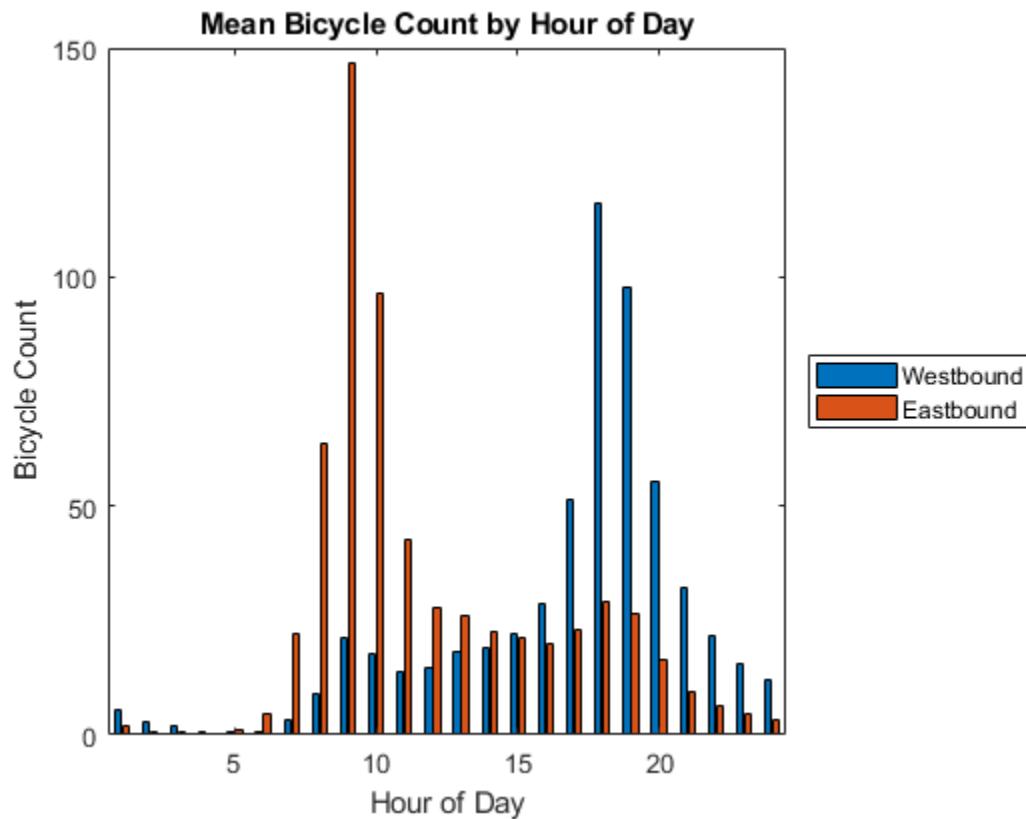
The bar plot indicates that traffic is heavier on weekdays. Also, there is a difference in the Eastbound and Westbound directions. This might indicate that people tend to take different routes when entering and leaving the city. Another possibility is that some people enter on one day and return on another day.

Determine the hour of day and use varfun for calculations by group.

```
bikeData.HrOfDay = hour(bikeData.Time);
byHr = varfun(@mean,bikeData(:,{'Westbound', 'Eastbound', 'HrOfDay'}),...
    'GroupingVariables','HrOfDay', 'OutputFormat', 'table');
head(byHr)
```

HrOfDay	GroupCount	mean_Westbound	mean_Eastbound
0	389	5.4396	1.7686
1	389	2.7712	0.87147
2	391	1.8696	0.58312
3	391	0.7468	0.289
4	391	0.52685	1.0026
5	391	0.70588	4.7494
6	391	3.1228	22.097
7	391	9.1176	63.54

```
bar(byHr{:, {'mean_Westbound', 'mean_Eastbound'}})
legend('Westbound', 'Eastbound', 'Location', 'eastoutside')
xlabel('Hour of Day')
ylabel('Bicycle Count')
title('Mean Bicycle Count by Hour of Day')
```



There are traffic spikes at the typical commute hours, around 9:00 a.m. and 5:00 p.m. Also, the trends between the Eastbound and Westbound directions are different. In general, the Westbound direction is toward residential areas surrounding the Cambridge area and toward the Universities. The Eastbound direction is toward Boston.

The traffic is heavier later in the day in the Westbound direction compared to the Eastbound direction. This might indicate university schedules and traffic due to restaurants in the area. Examine the trend by day of week as well as hour of day.

```
byHrDay = varfun(@sum,bikeData,'GroupingVariables',{'HrOfDay','Day'},...
    'OutputFormat','table');
head(byHrDay)

ans=8×6 table
  HrOfDay      Day       GroupCount    sum_Total    sum_Westbound   sum_Eastbound
  ____        _____    _____      _____      _____      _____
  0           Sunday     56          473         345          128
  0           Monday     55          202         145          57
  0           Tuesday    55          297         213          84
  0           Wednesday  56          374         286          88
```

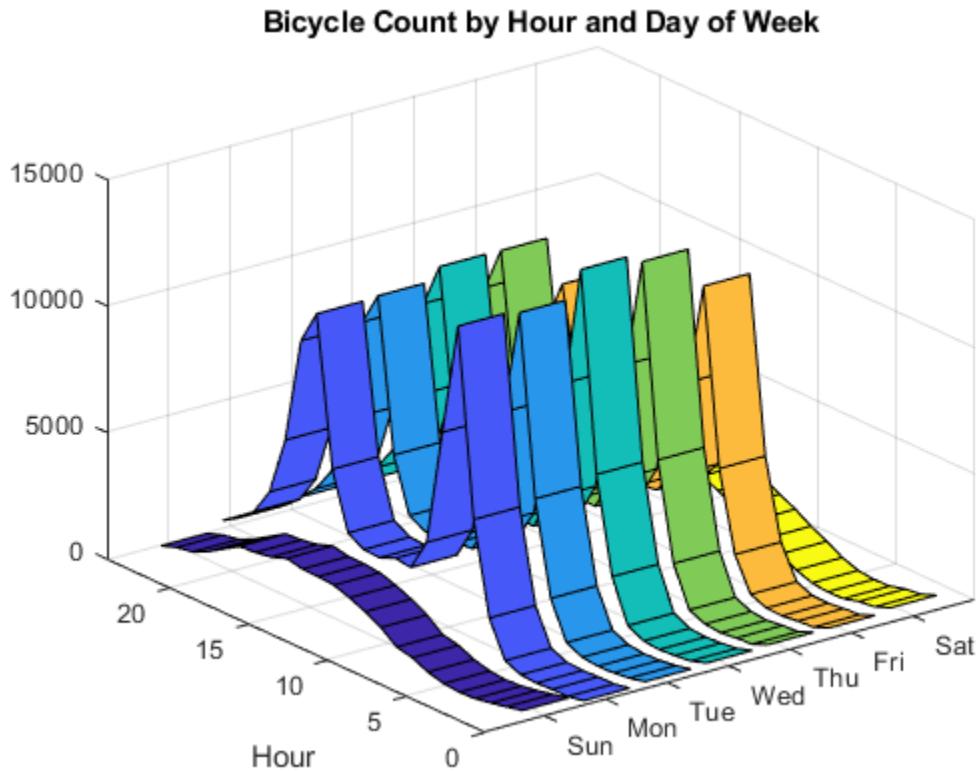
0	Thursday	56	436	324	112
0	Friday	55	442	348	94
0	Saturday	56	580	455	125
1	Sunday	56	333	259	74

To arrange the timetable so that the days of the week are the variables, use the `unstack` function.

```
hrAndDayWeek = unstack(byHrDay(:, {'HrOfDay', 'Day', 'sum_Total'}), 'sum_Total', 'Day');
head(hrAndDayWeek)
```

HrOfDay	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
0	473	202	297	374	436	442	580
1	333	81	147	168	173	183	332
2	198	77	68	93	128	141	254
3	86	41	43	44	50	61	80
4	51	81	117	101	108	80	60
5	105	353	407	419	381	340	128
6	275	1750	1867	2066	1927	1625	351
7	553	5355	5515	5818	5731	4733	704

```
ribbon(hrAndDayWeek.HrOfDay, hrAndDayWeek(:, 2:end))
ylim([0 24])
xlim([0 8])
xticks(1:7)
xticklabels({'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'})
ylabel('Hour')
title('Bicycle Count by Hour and Day of Week')
```



There are similar trends for the regular work days of Monday through Friday, with peaks at rush hour and traffic tapering off in the evening. Friday has less volume, though the overall trend is similar to the other work days. The trends for Saturday and Sunday are similar to each other, without rush hour peaks and with more volume later in the day. The late evening trends are also similar for Monday through Friday, with less volume on Friday.

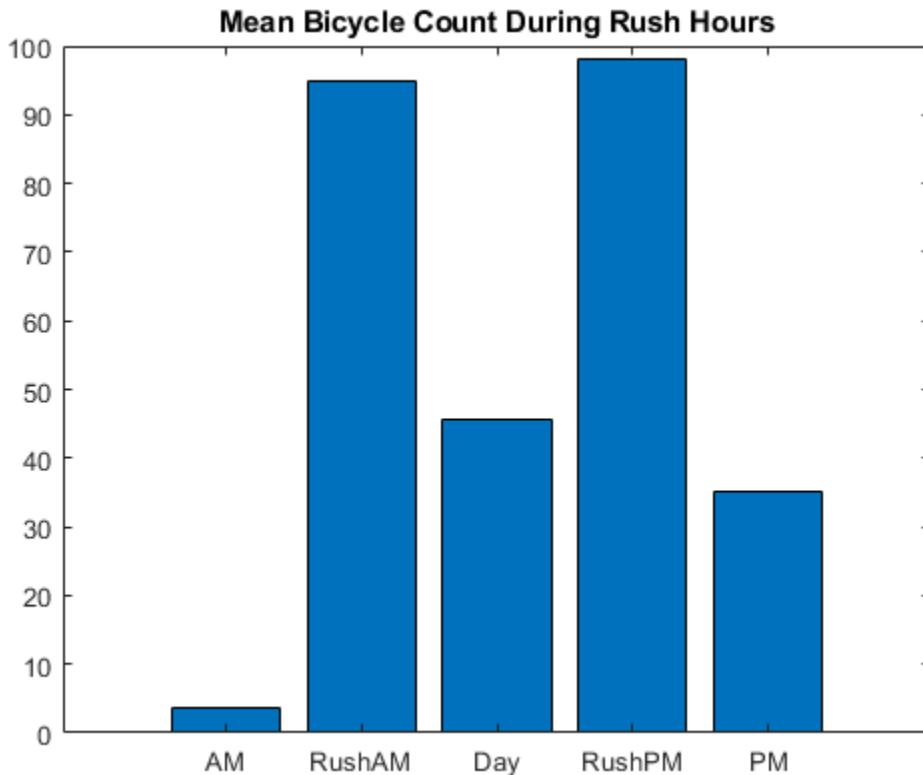
Analyze Traffic During Rush Hour

To examine the overall time of day trends, split up the data by rush hour times. It is possible to use different times of day or units of time using the `discretize` function. For example, separate the data into groups for AM, AMRush, Day, PMRush, PM. Then use `varfun` to calculate the mean by group.

```
bikeData.HrLabel = discretize(bikeData.HrOfDay,[0,6,10,15,19,24], 'categorical',...
    {'AM','RushAM','Day','RushPM','PM'});
byHrBin = varfun(@mean,bikeData(:,{'Total','HrLabel'}), 'GroupingVariables', 'HrLabel',...
    'OutputFormat', 'table')
```

HrLabel	GroupCount	mean_Total
AM	2342	3.5508
RushAM	1564	94.893
Day	1955	45.612
RushPM	1564	98.066
PM	1955	35.198

```
bar(byHrBin.mean_Total)
cats = categories(byHrBin.HrLabel);
xticklabels(cats)
title('Mean Bicycle Count During Rush Hours')
```



In general, there is about twice as much traffic in this area during the evening and morning rush hours compared to other times of the day. There is very little traffic in this area in the early morning, but there is still significant traffic in the evening and late evening, comparable to the day outside of the morning and evening rush hours.

See Also

[datetime](#) | [head](#) | [retime](#) | [rmmissing](#) | [sortrows](#) | [summary](#) | [table2timetable](#) | [timerange](#) | [timetable](#) | [unstack](#) | [varfun](#)

Related Examples

- “Represent Dates and Times in MATLAB” on page 7-2
- “Create Timetables” on page 10-2
- “Resample and Aggregate Data in Timetable” on page 10-5
- “Clean Timetable with Missing, Duplicate, or Nonuniform Times” on page 10-27
- “Select Times in Timetable” on page 10-19

Structures

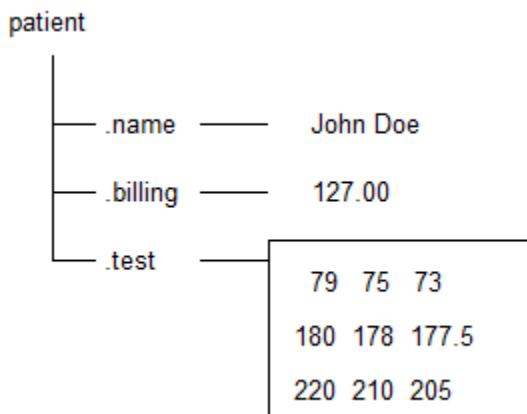
- “Structure Arrays” on page 11-2
- “Concatenate Structures” on page 11-8
- “Generate Field Names from Variables” on page 11-10
- “Access Data in Nested Structures” on page 11-11
- “Access Elements of a Nonscalar Structure Array” on page 11-13
- “Ways to Organize Data in Structure Arrays” on page 11-15
- “Memory Requirements for Structure Array” on page 11-18

Structure Arrays

When you have data that you want to organize by name, you can use structures to store it. Structures store data in containers called *fields*, which you can then access by the names you specify. Use dot notation to create, assign, and access data in structure fields. If the value stored in a field is an array, then you can use array indexing to access elements of the array. When you store multiple structures as a structure array, you can use array indexing and dot notation to access individual structures and their fields.

Create Scalar Structure

First, create a structure named `patient` that has fields storing data about a patient. The diagram shows how the structure stores data. A structure like `patient` is also referred to as a *scalar structure* because the variable stores one structure.



Use dot notation to add the fields `name`, `billing`, and `test`, assigning data to each field. In this example, the syntax `patient.name` creates both the structure and its first field. The commands that follow add more fields.

```
patient.name = 'John Doe';
patient.billing = 127;
patient.test = [79 75 73; 180 178 177.5; 220 210 205]

patient = struct with fields:
    name: 'John Doe'
    billing: 127
    test: [3x3 double]
```

Access Values in Fields

After you create a field, you can keep using dot notation to access and change the value it stores.

For example, change the value of the `billing` field.

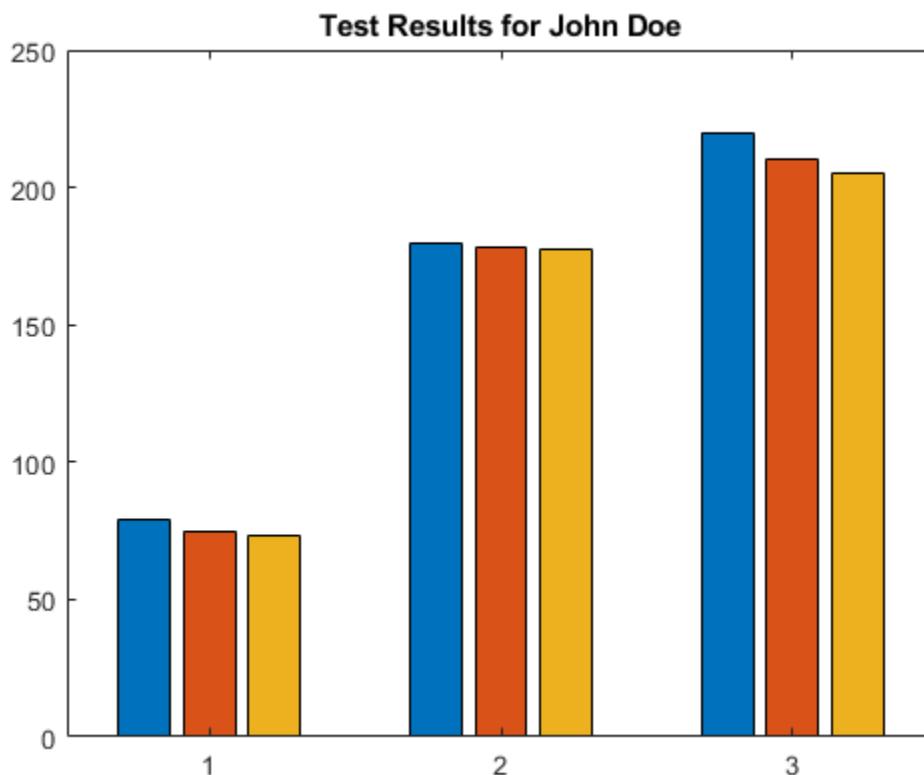
```
patient.billing = 512.00

patient = struct with fields:
    name: 'John Doe'
```

```
billing: 512
test: [3x3 double]
```

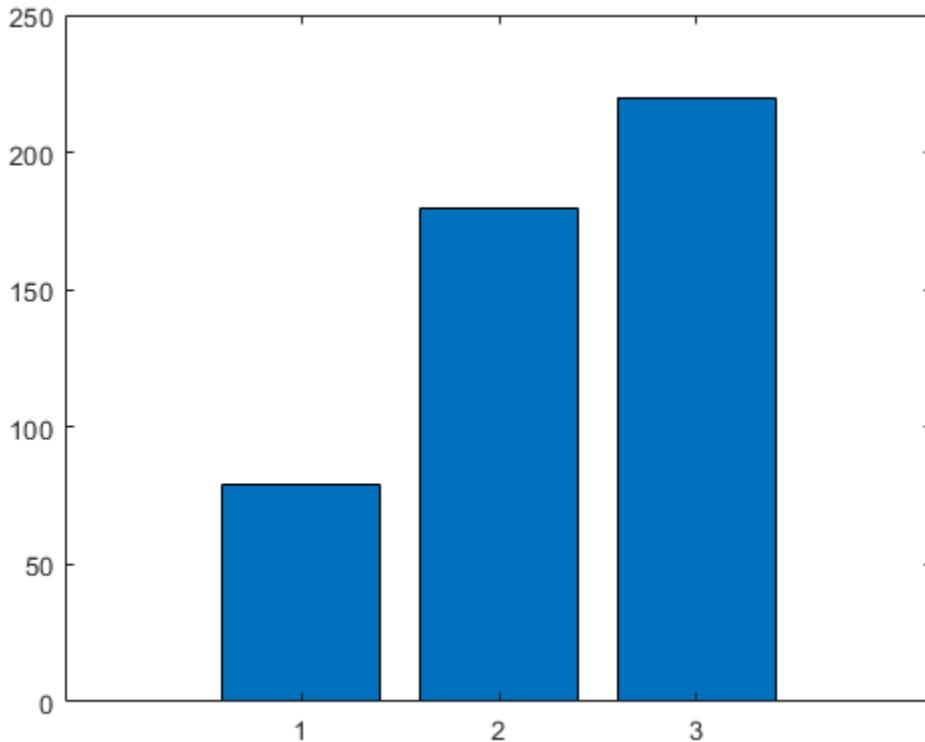
With dot notation, you also can access the value of any field. For example, make a bar chart of the values in `patient.test`. Add a title with the text in `patient.name`. If a field stores an array, then this syntax returns the whole array.

```
bar(patient.test)
title("Test Results for " + patient.name)
```



To access part of an array stored in a field, add indices that are appropriate for the size and type of the array. For example, create a bar chart of the data in one column of `patient.test`.

```
bar(patient.test(:,1))
```



Index into Nonscalar Structure Array

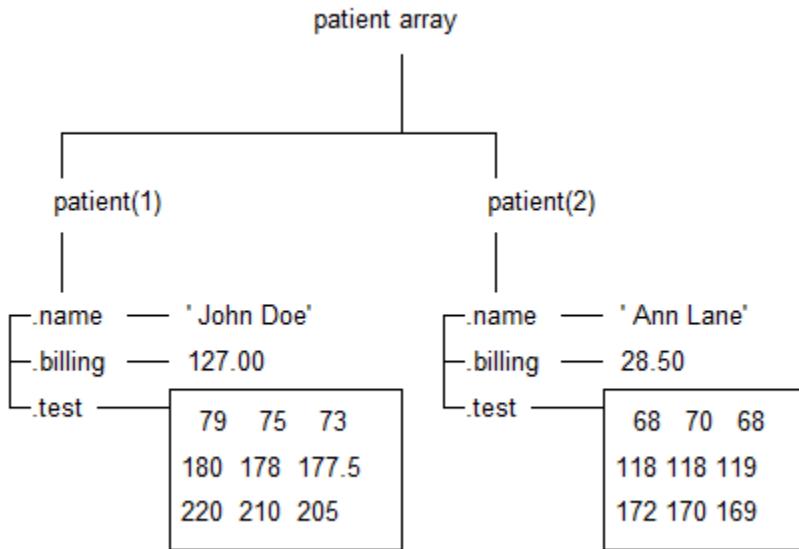
Structure arrays can be nonscalar. You can create a structure array having any size, as long as each structure in the array has the same fields.

For example, add a second structure to `patients` having data about a second patient. Also, assign the original value of 127 to the `billing` field of the first structure. Since the array now has two structures, you must access the first structure by indexing, as in `patient(1).billing = 127`.

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];
patient(1).billing = 127

patient=1x2 struct array with fields:
  name
  billing
  test
```

As a result, `patient` is a 1-by-2 structure array with contents shown in the diagram.



Each patient record in the array is a structure of class `struct`. An array of structures is sometimes referred to as a *struct array*. However, the terms *struct array* and *structure array* mean the same thing. Like other MATLAB® arrays, a structure array can have any dimensions.

A structure array has the following properties:

- All structures in the array have the same number of fields.
- All structures have the same field names.
- Fields of the same name in different structures can contain different types or sizes of data.

If you add a new structure to the array without specifying all of its fields, then the unspecified fields contain empty arrays.

```

patient(3).name = 'New Name';
patient(3)

ans = struct with fields:
  name: 'New Name'
  billing: []
  test: []
  
```

To index into a structure array, use array indexing. For example, `patient(2)` returns the second structure.

```

patient(2)

ans = struct with fields:
  name: 'Ann Lane'
  billing: 28.5000
  test: [3x3 double]
  
```

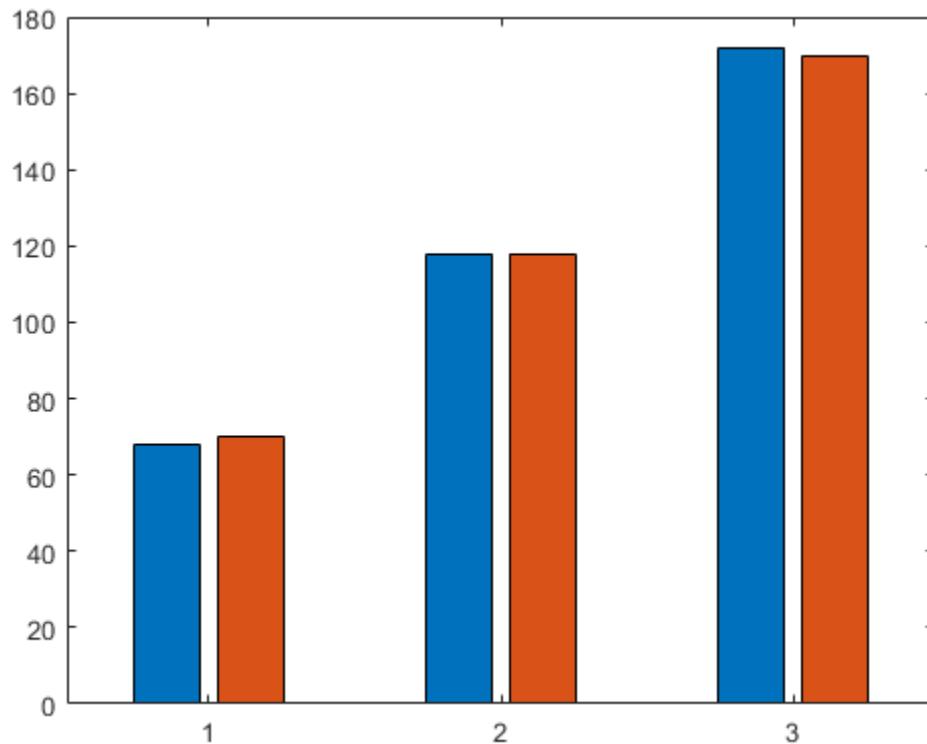
To access a field, use array indexing and dot notation. For example, return the value of the `billing` field for the second patient.

```
patient(2).billing
```

```
ans = 28.5000
```

You also can index into an array stored by a field. Create a bar chart displaying only the first two columns of `patient(2).test`.

```
bar(patient(2).test(:,[1 2]))
```



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 `patient(1:2).test(1:2,2:3)`, which attempt to index into a field for multiple elements of the structure array. Instead, use the `arrayfun` function.

See Also

`fieldnames` | `isfield` | `struct`

Related Examples

- “Access Elements of a Nonscalar Structure Array” on page 11-13
- “Generate Field Names from Variables” on page 11-10
- “Create Cell Array” on page 12-3
- “Cell vs. Structure Arrays” on page 12-16

- “Create and Work with Tables” on page 9-2
- “Advantages of Using Tables” on page 9-58

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=1×2 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=3×2 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”
- “Structure Arrays” on page 11-2
- “Access Elements of a Nonscalar Structure Array” on page 11-13

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 fieldnames*, or sometimes *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 fieldname 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.

See Also

[fieldnames](#) | [getfield](#) | [setfield](#) | [struct](#)

Related Examples

- “Variable Names” on page 1-4
- “Structure Arrays” on page 11-2

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
```

See Also

[getfield](#) | [setfield](#) | [struct](#)

Related Examples

- “Structure Arrays” on page 11-2
- “Ways to Organize Data in Structure Arrays” on page 11-15

Access Elements of a Nonscalar Structure 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 structure 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 2-79
- “Anonymous Functions” on page 20-20

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 11-15
- “Element-by-Element Organization” on page 11-16

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.

Plane Organization

Consider an RGB image with three arrays corresponding to color intensity values.

	Blue intensity values	<table border="1"> <tbody> <tr><td>0.689</td><td>0.706</td><td>0.118</td><td>0.884</td><td>...</td></tr> <tr><td>0.535</td><td>0.532</td><td>0.653</td><td>0.925</td><td>...</td></tr> <tr><td>0.314</td><td>0.265</td><td>0.159</td><td>0.101</td><td>...</td></tr> <tr><td>0.553</td><td>0.633</td><td>0.528</td><td>0.493</td><td>...</td></tr> <tr><td>0.441</td><td>0.465</td><td>0.512</td><td>0.512</td><td>...</td></tr> <tr><td>0.398</td><td>0.401</td><td>0.421</td><td>0.398</td><td>...</td></tr> </tbody> </table>	0.689	0.706	0.118	0.884	...	0.535	0.532	0.653	0.925	...	0.314	0.265	0.159	0.101	...	0.553	0.633	0.528	0.493	...	0.441	0.465	0.512	0.512	...	0.398	0.401	0.421	0.398	...																																		
0.689	0.706	0.118	0.884	...																																																														
0.535	0.532	0.653	0.925	...																																																														
0.314	0.265	0.159	0.101	...																																																														
0.553	0.633	0.528	0.493	...																																																														
0.441	0.465	0.512	0.512	...																																																														
0.398	0.401	0.421	0.398	...																																																														
	Green intensity values	<table border="1"> <tbody> <tr><td>0.342</td><td>0.647</td><td>0.515</td><td>0.816</td><td>...</td></tr> <tr><td>0.111</td><td>0.300</td><td>0.205</td><td>0.526</td><td>...</td></tr> <tr><td>0.523</td><td>0.428</td><td>0.712</td><td>0.929</td><td>...</td></tr> <tr><td>0.214</td><td>0.604</td><td>0.918</td><td>0.344</td><td>...</td></tr> <tr><td>0.100</td><td>0.121</td><td>0.113</td><td>0.126</td><td>...</td></tr> <tr><td>0.288</td><td>0.187</td><td>0.204</td><td>0.175</td><td>...</td></tr> </tbody> </table>	0.342	0.647	0.515	0.816	...	0.111	0.300	0.205	0.526	...	0.523	0.428	0.712	0.929	...	0.214	0.604	0.918	0.344	...	0.100	0.121	0.113	0.126	...	0.288	0.187	0.204	0.175	...																																		
0.342	0.647	0.515	0.816	...																																																														
0.111	0.300	0.205	0.526	...																																																														
0.523	0.428	0.712	0.929	...																																																														
0.214	0.604	0.918	0.344	...																																																														
0.100	0.121	0.113	0.126	...																																																														
0.288	0.187	0.204	0.175	...																																																														
Red intensity values	<table border="1"> <tbody> <tr><td>0.112</td><td>0.986</td><td>0.234</td><td>0.432</td><td>...</td></tr> <tr><td>0.765</td><td>0.128</td><td>0.863</td><td>0.521</td><td>...</td></tr> <tr><td>1.000</td><td>0.985</td><td>0.761</td><td>0.698</td><td>...</td></tr> <tr><td>0.455</td><td>0.783</td><td>0.224</td><td>0.395</td><td>...</td></tr> <tr><td>0.021</td><td>0.500</td><td>0.311</td><td>0.123</td><td>...</td></tr> <tr><td>1.000</td><td>1.000</td><td>0.867</td><td>0.051</td><td>...</td></tr> <tr><td>1.000</td><td>0.945</td><td>0.998</td><td>0.893</td><td>...</td></tr> <tr><td>0.990</td><td>0.941</td><td>1.000</td><td>0.876</td><td>...</td></tr> <tr><td>0.902</td><td>0.867</td><td>0.834</td><td>0.798</td><td>...</td></tr> <tr><td>.</td><td></td><td></td><td></td><td></td></tr> <tr><td>.</td><td></td><td></td><td></td><td></td></tr> </tbody> </table>	0.112	0.986	0.234	0.432	...	0.765	0.128	0.863	0.521	...	1.000	0.985	0.761	0.698	...	0.455	0.783	0.224	0.395	...	0.021	0.500	0.311	0.123	...	1.000	1.000	0.867	0.051	...	1.000	0.945	0.998	0.893	...	0.990	0.941	1.000	0.876	...	0.902	0.867	0.834	0.798					<table border="1"> <tbody> <tr><td>912</td><td>0.713</td><td>...</td></tr> <tr><td>219</td><td>0.328</td><td>...</td></tr> <tr><td>128</td><td>0.133</td><td>...</td></tr> </tbody> </table>	912	0.713	...	219	0.328	...	128	0.133	...
0.112	0.986	0.234	0.432	...																																																														
0.765	0.128	0.863	0.521	...																																																														
1.000	0.985	0.761	0.698	...																																																														
0.455	0.783	0.224	0.395	...																																																														
0.021	0.500	0.311	0.123	...																																																														
1.000	1.000	0.867	0.051	...																																																														
1.000	0.945	0.998	0.893	...																																																														
0.990	0.941	1.000	0.876	...																																																														
0.902	0.867	0.834	0.798	...																																																														
.																																																																		
.																																																																		
912	0.713	...																																																																
219	0.328	...																																																																
128	0.133	...																																																																

If you have arrays RED, GREEN, and BLUE in your workspace, then these commands create a scalar structure named img that uses plane organization:

```
img.red = RED;
img.green = GREEN;
img.blue = BLUE;
```

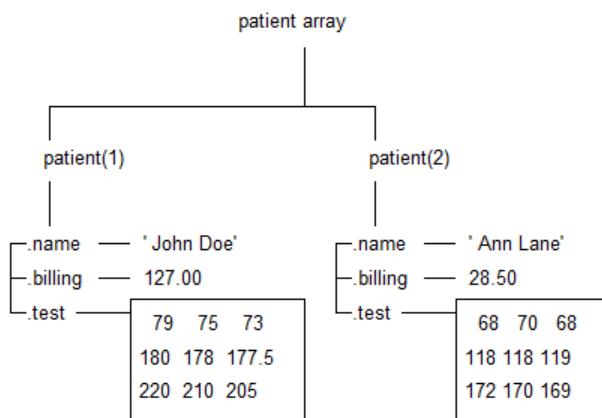
Plane organization allows you to easily extract entire image planes for display, filtering, or other processing. For example, multiply the red intensity values by 0.9:

```
adjustedRed = .9 * img.red;
```

If you have multiple images, you can add them to the `img` structure, so that each element `img(1), ..., img(n)` contains an entire image. For an example that adds elements to a structure, see the following section.

Element-by-Element Organization

Consider a database with patient information. Each record contains data for the patient's name, test results, and billing amount.



These statements create an element in a structure array named `patient`:

```
patient(1).name = 'John Doe';
patient(1).billing = 127.00;
patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
```

Additional patients correspond to new elements in the structure. For example, add an element for a second patient:

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
```

Element-by-element organization supports simple indexing to access data for a particular patient. For example, find the average of the first patient's test results, calculating by rows (dimension 2) rather than by columns:

```
aveResultsDoe = mean(patient(1).test,2)
```

This code returns

```
aveResultsDoe =
75.6667
178.5000
212.0000
```

See Also

`struct`

More About

- “Structure Arrays” on page 11-2
- “Access Elements of a Nonscalar Structure Array” on page 11-13
- “Memory Requirements for Structure Array” on page 11-18

Memory Requirements for 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.

Preallocate 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 30-10

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 12-9
- “Combine Cell Arrays” on page 12-10
- “Pass Contents of Cell Arrays to Functions” on page 12-11
- “Preallocate Memory for Cell Array” on page 12-15
- “Cell vs. Structure Arrays” on page 12-16
- “Multilevel Indexing to Access Parts of Cells” on page 12-20

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=2×3 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

- “Structure Arrays” on page 11-2
- “Create and Work with Tables” on page 9-2
- “Cell vs. Structure Arrays” on page 12-16
- “Advantages of Using Tables” on page 9-58

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=2×3 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=2×2 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=2×3 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=1×3 cell array
 {[1]}    {[2]}    {[3]}

numericVector = cell2mat(numericCells)

numericVector = 1×3

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 = 1x3
```

```
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 12-20
- “Comma-Separated Lists” on page 2-79

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=1×3 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=4×4 cell array
  {[      1]}    {[      2]}    {[      3]}    {[      4]}    {[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=5×5 cell array
 Columns 1 through 4
  {[      1]}    {[      2]}    {[      3]}    {[      4]}    {[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 12-10
- “Delete Data from Cell Array” on page 12-9

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=3×3 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=3×3 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=2×3 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”

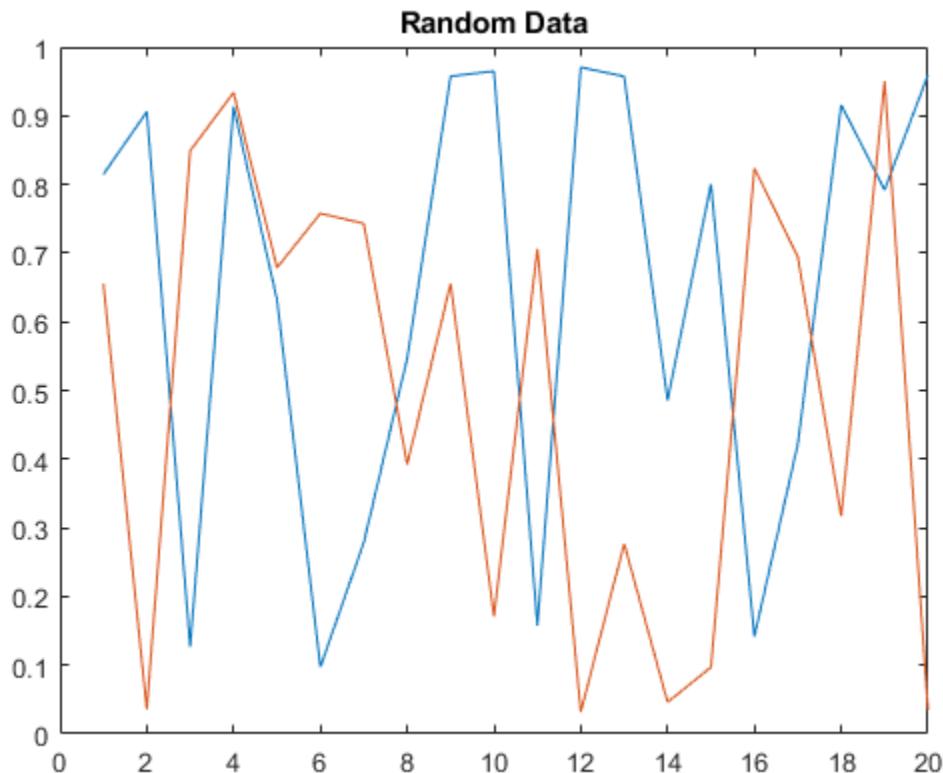
Pass Contents of Cell Arrays to Functions

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 Contents of Single Cell by Indexing with Curly Braces, {}

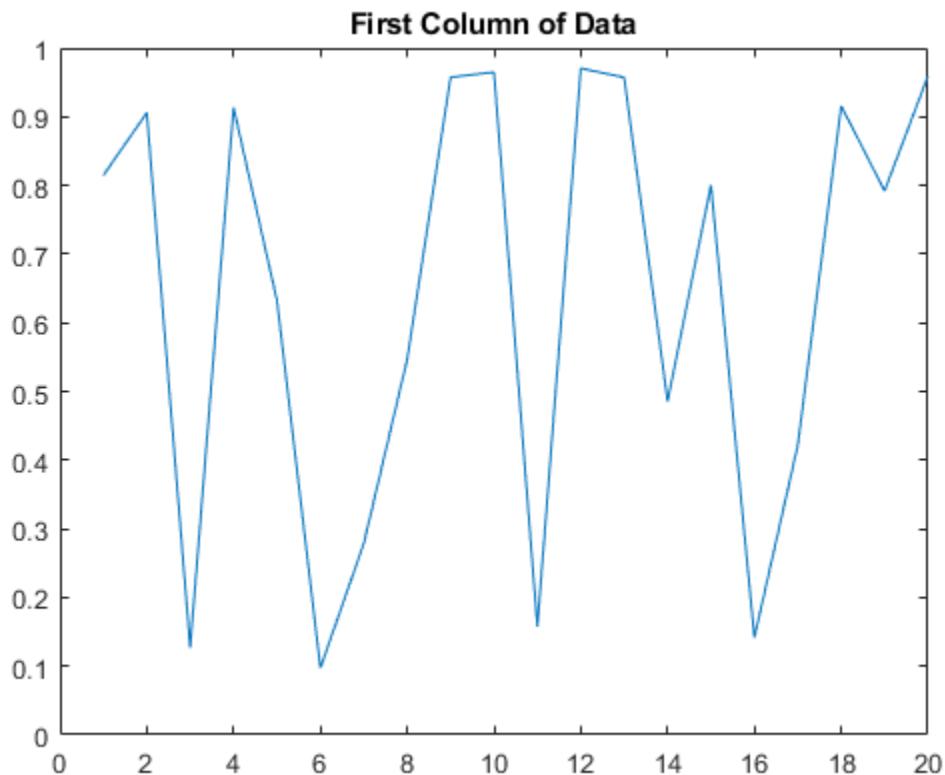
This example creates a cell array that contains text and a 20-by-2 array of random numbers.

```
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).

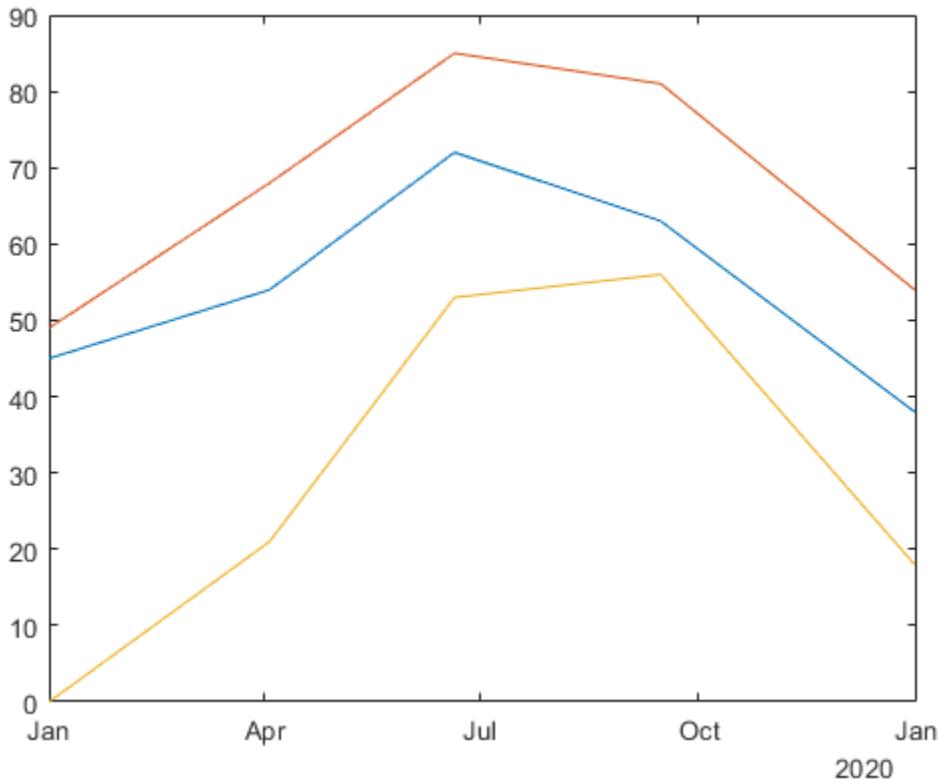
```
figure
plot(randCell{1,2}{:,1})
title('First Column of Data')
```



Combine Numeric Data from Multiple Cells Using `cell2mat`

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,:) = {'2020-01-01', [45, 49, 0]};  
temperature(2,:) = {'2020-04-03', [54, 68, 21]};  
temperature(3,:) = {'2020-06-20', [72, 85, 53]};  
temperature(4,:) = {'2020-09-15', [63, 81, 56]};  
temperature(5,:) = {'2020-12-31', [38, 54, 18]};  
  
allTemps = cell2mat(temperature(:,2));  
dates = datetime(temperature(:,1));  
  
plot(dates, allTemps)
```



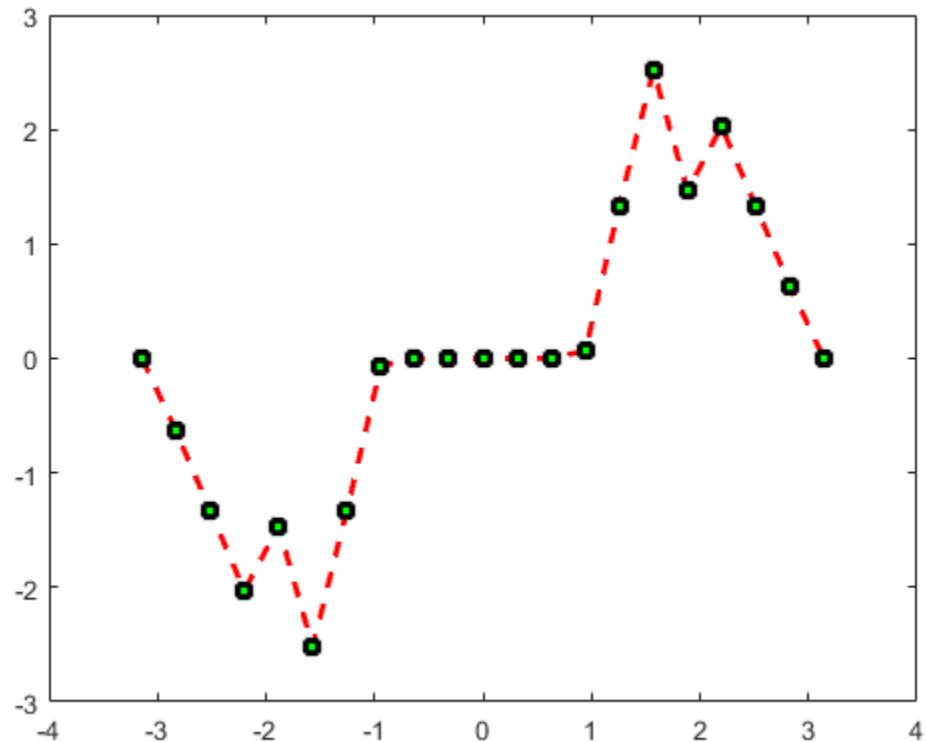
Pass Contents of Multiple Cells as Comma Separated List To Function

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 12-20
- “Comma-Separated Lists” on page 2-79

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 30-10

Cell vs. Structure 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];

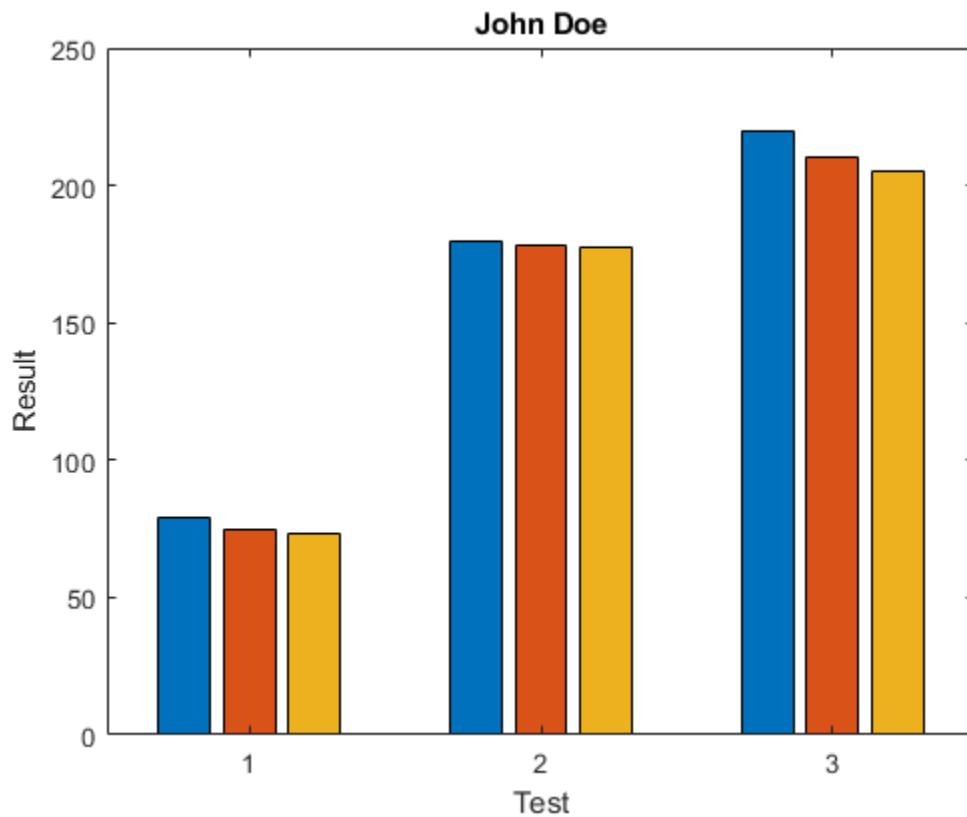
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];

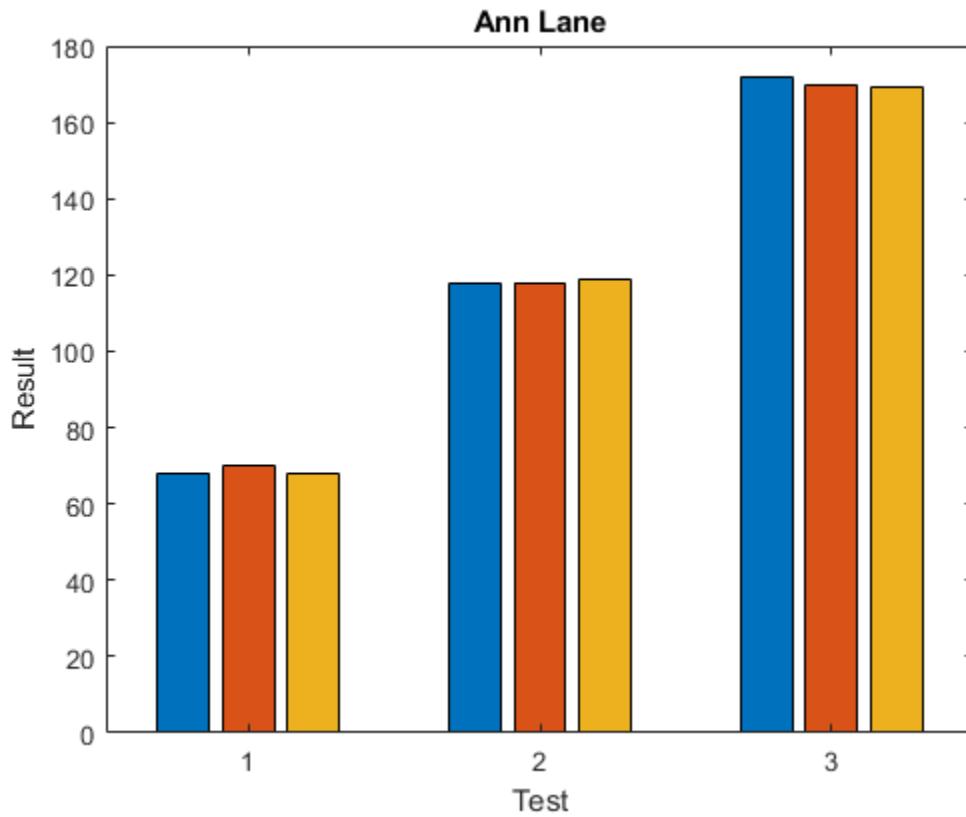
patient

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.

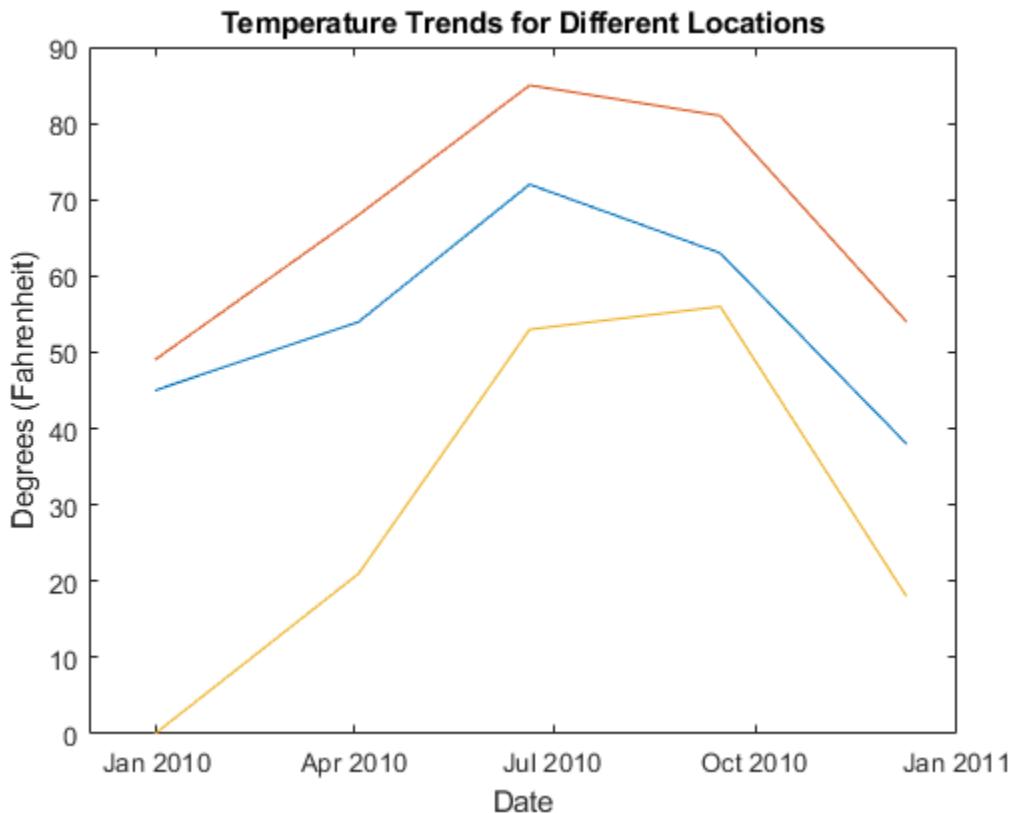
For example, store temperature data for three cities over time in a cell array.

```
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=5×2 cell array  
{'2009-12-31'} {[ 45 49 0]}  
{'2010-04-03'} {[54 68 21]}  
{'2010-06-20'} {[72 85 53]}  
{'2010-09-15'} {[63 81 56]}  
{'2010-12-09'} {[38 54 18]}
```

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
- “Structure Arrays” on page 11-2
- “Access Data in Tables” on page 9-34

More About

- “Advantages of Using Tables” on page 9-58

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=2×2 cell array
  {[ 1 2 3]}    {[100 200 300]}
  {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 = 1×3
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}(1,2)` 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}{1,2}` 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 13-5
- “Call Local Functions Using Function Handles” on page 13-6
- “Compare Function Handles” on page 13-8

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 13-3
- "Arrays of Function Handles" on page 13-4
- "Saving and Loading Function Handles" on page 13-4

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:

- Passing a function to another function (often called *function functions*). For example, passing a function to integration and optimization functions, such as `integral` and `fzero`.
- Specifying callback functions (for example, a callback that responds to a UI event or interacts with data acquisition hardware).
- Constructing handles to functions defined inline instead of stored in a program file (anonymous functions).
- Calling 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 =
```

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 20-37.
- Overloading — When a function handle is invoked with one or more arguments, MATLAB determines the dominant argument. If the dominant argument is an object, MATLAB determines if the object’s class has a method which overloads the same name as the function handle’s associated function. If it does, then the object’s method is invoked instead of the associated function.

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 20-20.

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` | `function_handle` | `functions` | `isa` | `str2func`

Related Examples

- “Pass Function to Another Function” on page 13-5

More About

- “Anonymous Functions” on page 20-20

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 $[0, \text{Inf}]$:

```
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 20-20

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 20-25

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 14-8
- “Read and Write Using Key Index” on page 14-9
- “Modify Keys and Values in Map” on page 14-13
- “Map to Different Value Types” on page 14-15

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	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Aug → 37.3

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 14-8

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 <code>Map</code> contains specified key
<code>keys</code>	Names of all keys in <code>Map</code>
<code>length</code>	Length of <code>Map</code>
<code>remove</code>	Remove key and its value from <code>Map</code>
<code>size</code>	Dimensions of <code>Map</code>
<code>values</code>	Values contained in <code>Map</code>

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 14-8

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 14-4

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(...  
    {'keystr1', 'keystr2', ...}, {val1, val2, ...});
```

As an example of constructing an initialized `Map` object, create a new `Map` for the following key/value pairs taken from the monthly rainfall map shown earlier in this section.

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

```
k = {'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', ...
    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', 'Annual'};
```

```
v = {327.2, 368.2, 197.6, 178.4, 100.0, 69.9, ...
    32.3, 37.3, 19.0, 37.0, 73.2, 110.9, 1551.0};
```

```
rainfallMap = containers.Map(k, v)
```

```
rainfallMap =
```

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 14-10, 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 14-8

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 14-9
- “Modify Keys and Values in Map” on page 14-13
- “Map to Different Value Types” on page 14-15

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 = map0bj(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 =
```

```
6
```

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 14-8
- “Modify Keys and Values in Map” on page 14-13

- “Map to Different Value Types” on page 14-15

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 14-14, 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 14-8
- “Read and Write Using Key Index” on page 14-9
- “Map to Different Value Types” on page 14-15

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
- “Structure Arrays” on page 11-2
- “Create Cell Array” on page 12-3
- “Examine Contents of Map” on page 14-8
- “Read and Write Using Key Index” on page 14-9
- “Modify Keys and Values in Map” on page 14-13

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 almost 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
- “Concatenating Objects of Different Classes”

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 leftmost 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(10000000)]
A =
1x3 int16 row vector
450      250     32767
```

Example of Combining Unlike Integer Sizes

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 leftmost type determines the data type for all elements in the vector:

```
A = [int16(5000) int8(50)]
A =
1x2 int16 row vector
5000      50
B = [int8(50) int16(5000)]
B =
1x2 int8 row vector
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 =
2x1 int8 column vector
```

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 leftmost element determines the data type for all elements in the resulting matrix:

```
A = [int8(-100) uint8(100)]
```

A =

1×2 int8 row vector

-100 100

```
B = [uint8(100) int8(-100)]
```

B =

1×2 uint8 row vector

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 =

1×2 int8 row vector

50 0

See Also

More About

- “Integers” on page 4-2
- “Integer Arithmetic” on page 4-19

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 15-8
- “Combining Logical and Double Types” on page 15-8

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 16-5

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
```

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 18-3
- “Code Sections” on page 18-5
- “Scripts vs. Functions” on page 18-12
- “Add Functions to Scripts” on page 18-14

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

- “Code Sections” on page 18-5
- “Scripts vs. Functions” on page 18-12
- “Base and Function Workspaces” on page 20-9
- “Create Live Scripts in the Live Editor” on page 19-6

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 19-6.

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 Editor and Live Editor include 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** or **Live Editor** tab, and press the  button. The comment wraps to the next line:

```
% This is a program that has a comment that is a little more than 75  
% columns wide.  
disp('Hello, world')
```

By default, as you type comments in the Editor and Live 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. To adjust **Comment formatting** preferences in MATLAB Online, select **Editor/Debugger > MATLAB Language**.

The Editor and Live Editor do 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

See Also

Related Examples

- “Add Help for Your Program” on page 20-5
- “Create Scripts” on page 18-2
- “Create Live Scripts in the Live Editor” on page 19-6
- “Editor/Debugger Preferences”

Code Sections

In this section...

- “Divide Your File into Code Sections” on page 18-5
- “Evaluate Code Sections” on page 18-5
- “Navigate Among Code Sections in a File” on page 18-6
- “Example of Evaluating Code Sections” on page 18-7
- “Change the Appearance of Code Sections” on page 18-9
- “Use Code Sections with Control Statements and Functions” on page 18-9

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, click  (or  **Section Break** in MATLAB Online).
- 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.	<p>With the cursor in the code section, on the Editor tab, in the Run section, click  Run Section.</p> <p>In MATLAB Online, the  Run Section button is located in the Section section.</p>

Operation	Instructions
Run the code in the current section, and then move to the next section.	With the cursor in the code section, on the Editor tab, in the Run section, click  Run and Advance . In MATLAB Online, the  Run and Advance button is located in the Section section.
Run all the code in the file.	On the Editor tab, in the Run section, click  Run . You also can type the saved script name in the Command Window.

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.	On the Editor tab, in the Run section, click  Advance .
Move to the previous section.	Press Ctrl + Up arrow .

Operation	Instructions
Move to a specific section.	On the Editor tab, in the Navigate section, use the  Go To to move the cursor to a selected section.

In MATLAB Online, to navigate among sections, on the **Editor** tab, in the **Navigate** section, select  **Go To**. Then, select from the available options.

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
```

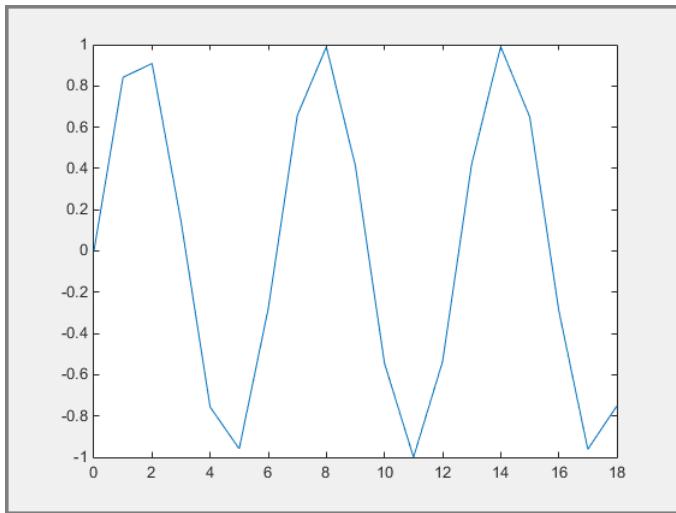
- 2 Insert a blank line and a second section break after `plot(x,y)`. Add a section title, **Modify Plot Properties**, so that the entire file contains this code:

```
%% 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';
```

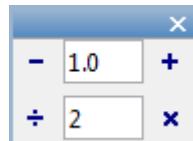
- 3 Save the file.
- 4 Place your cursor in the section titled **Calculate and Plot Sine Wave**. On the **Editor** tab, in the **Run** section, click  **Run Section**. In MATLAB Online, the  **Run Section** button is located in the **Section** section.

A figure displaying a course plot of $\sin(x)$ appears.



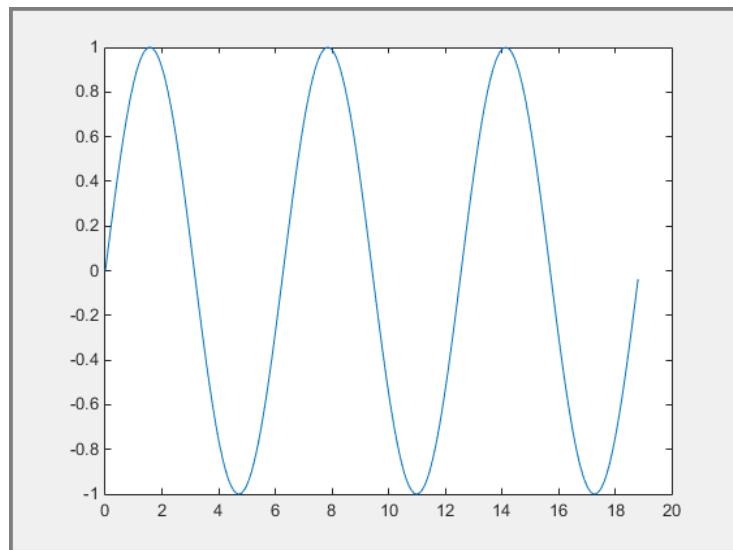
5 Smooth the sine plot.

- 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.

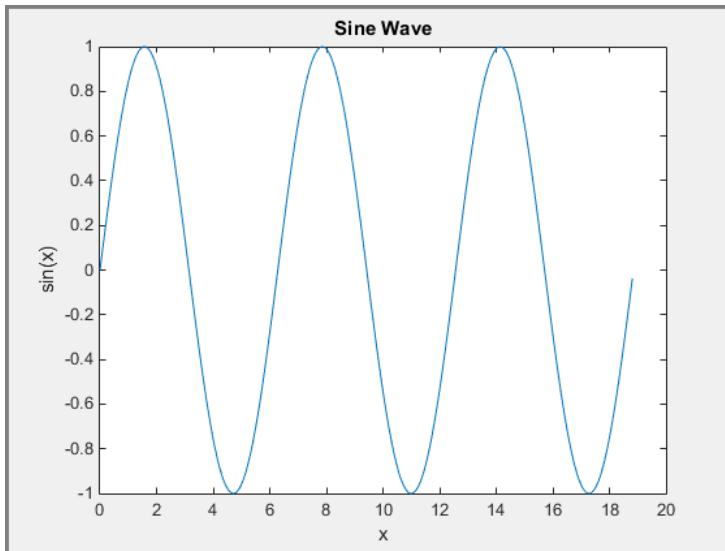


- 3 Type 2 in the \div / \times text box.
- 4 Click the \div button several times.

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 19-6
- “Scripts vs. Functions” on page 18-12

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 `triarea` 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 18-14
- “Base and Function Workspaces” on page 20-9

Add Functions to Scripts

Starting in R2016b, 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.

Create a Script with Local Functions

To create a script or live script with local functions, go to the **Home** tab and select **New > Script** or **New > Live Script**. Then, add code to the script. Add all local functions at end of the file, after the script code. Include at least one line of script code before the local functions. Each local function must begin with its own function definition statement and end with the `end` keyword. The functions can appear in any order.

For example, create a script called `mystats.m`.

```
edit mystats
```

In the file, include two local functions, `mymean` and `mymedian`. The script `mystats` declares an array, determines the length of the array, and then uses the local functions `mymean` and `mymedian` to calculate the average and median of the array.

```
x = 1:10;
n = length(x);
avg = mymean(x,n);
med = mymedian(x,n);

function a = mymean(v,n)
% MYMEAN Local function that calculates mean of array.

    a = sum(v)/n;
end

function m = mymedian(v,n)
% MYMEDIAN Local function that calculates median of array.

    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
```

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. This is because live scripts do not support individual sections within local functions.

Run Scripts with Local Functions

To run a script or live script that includes local functions, in the **Editor** or **Live Editor** tab, click the  **Run** button. You also can type the saved script name in the Command Window.

To run an individual section, place the cursor inside the section and use the  **Run Section** button (requires R2017b or later for .m files). In .mlx files, you can run sections only when they are before the local function definitions.

Restrictions for Local Functions and Variables

Local functions are only visible within the file where they are defined. They are not visible to functions in other files, and cannot be called from the Command Window.

Local functions in the current file have precedence over functions in other files. That is, when you call a function within a script, MATLAB checks whether the function is a local function before looking for the function in other locations. 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, like all 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.

Access Help for 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` command. Specify the names of both the script and the local function, separating them with a `>` character.

For example:

```
help mystats>mymean
mymean Local function that calculates mean of array.
```

See Also

More About

- “Create Functions in Files” on page 20-2
- “Function Precedence Order” on page 20-37
- “Base and Function Workspaces” on page 20-9

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.

Distribution of Fatalities

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

We can experiment with the data to see if any of the variables in the table are correlated with highway fatality rates. It appears that highway fatality rates are lower in states with a higher percentage urban population.

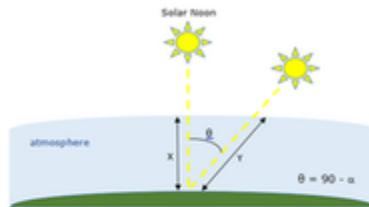
```
varName = 'urbanPopulation';
scatter(fatalities.(varName),rate)
 xlabel(varName)
 ylabel('Percent Fatalities per Million Vehicle Miles')

hold on
xmin = min(fatalities.(varName));
xmax = max(fatalities.(varName));
p = polyfit(fatalities.(varName),rate,1);
plot([xmin xmax], polyval(p,[xmin xmax]))
```

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, Microsoft Word, or LaTeX documents 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/m^2) 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.

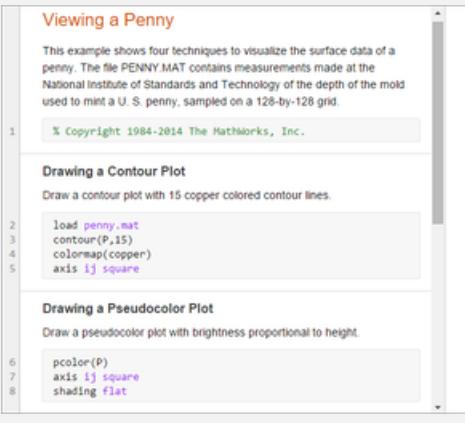
% Put your code here

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
Text Formatting	Add and view formatted text in the Live Editor	Use publishing markup to add formatted text, publish to view
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. 2 Drawing a Contour Plot 3 Draw a contour plot with 15 copper colored contour lines. 4 5 load penny.mat 6 contour(P,15) 7 colormap(copper) 8 axis ij square 9 10 Drawing a Pseudocolor Plot 11 Draw a pseudocolor plot with brightness proportional to height. 12 13 pcOLOR(P) 14 axis ij square 15 shading flat 16 17 18 19 20 21 22 23 24 25 </pre>	 <pre> 1 %Viewing a Penny 2 % This example shows four techniques to visualize the surface ... 3 % of a penny. The file PENNY.MAT contains measurements made at the ... 4 % National Institute of Standards and Technology of the depth of the ... 5 % mold used to mint a U. S. penny, sampled on a 128-by-128 ... 6 % grid. 7 8 % Copyright 1984-2014 The MathWorks, Inc. 9 %Drawing a Contour Plot 10 % Draw a contour plot with 15 copper colored contour lines. 11 12 load penny.mat 13 contour(P,15) 14 colormap(copper) 15 axis ij square 16 17 %Drawing a Pseudocolor Plot 18 % Draw a pseudocolor plot with brightness proportional to height. 19 20 pcOLOR(P) 21 axis ij square 22 shading flat 23 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 — Starting in R2019b, MATLAB supports the Live Editor in all operating systems supported by MATLAB. For more information, see System Requirements.

For MATLAB versions R2016a through R2019a, the Live Editor is not supported in several operating systems supported by MATLAB.

Unsupported operating systems 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.

In addition, some operating systems require additional configuration to run the Live Editor in MATLAB versions R2016a through R2019a. 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

Some MATLAB features are unsupported 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.
- MATLAB preferences — The Live Editor ignores some MATLAB preferences, including custom keyboard shortcuts and Emacs-style keyboard shortcuts.

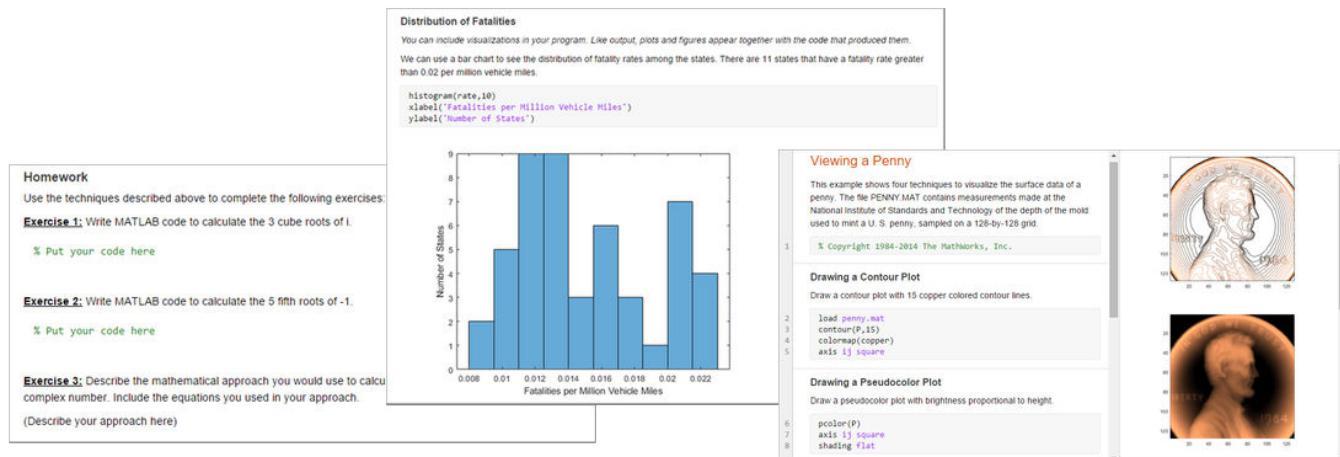
See Also

Related Examples

- “Create Live Scripts in the Live Editor” on page 19-6
- “Live Code File Format (`.mlx`)” on page 19-69
- MATLAB Live Script Gallery

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.



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**.

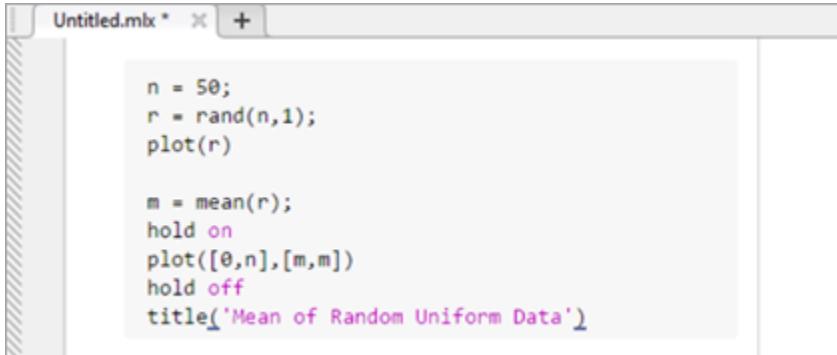
Note You must use one of the described conversion methods to convert your script to a live script. Simply renaming the script with a `.mlx` extension does not work and can corrupt the file.

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')
```



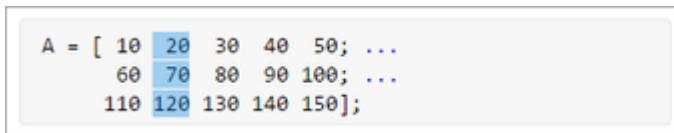
By default, MATLAB autocompletes block endings, parentheses, and quotes when entering code in the Live Editor. For example, type `if` and then press **Enter**. MATLAB automatically adds the `end` statement.



MATLAB also autocompletes comments, character vectors, strings, and parentheses 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.



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 or if MATLAB detects a significant issue in your code, 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.

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 change the size of the output display panel, drag the resizer bar between the code and output to the left or to the right.



To clear an output, right-click the output or the code line that created it, and select **Clear 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.

To open individual outputs, such as variables and figures, in a separate window, click the  button 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.

To modify figures in the output, use the tools in the upper-right corner of the figure axes or in the **Figure** toolbar. 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 19-23.

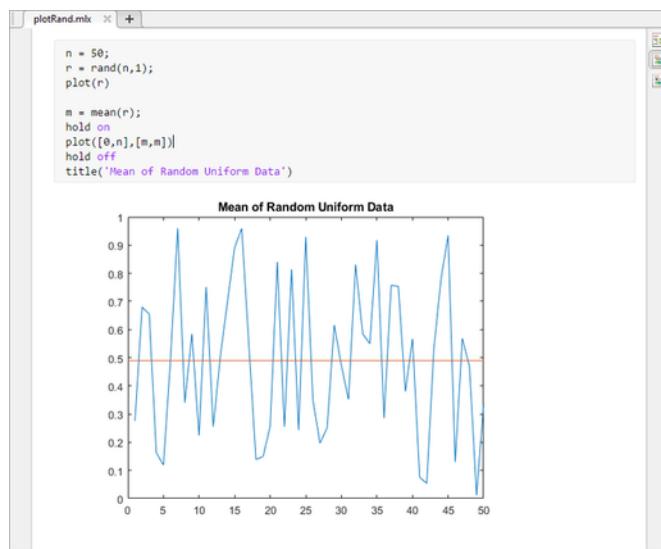
Change View

To optimize your live script for your current flow, you can change where to display output and whether to display code in the live script.

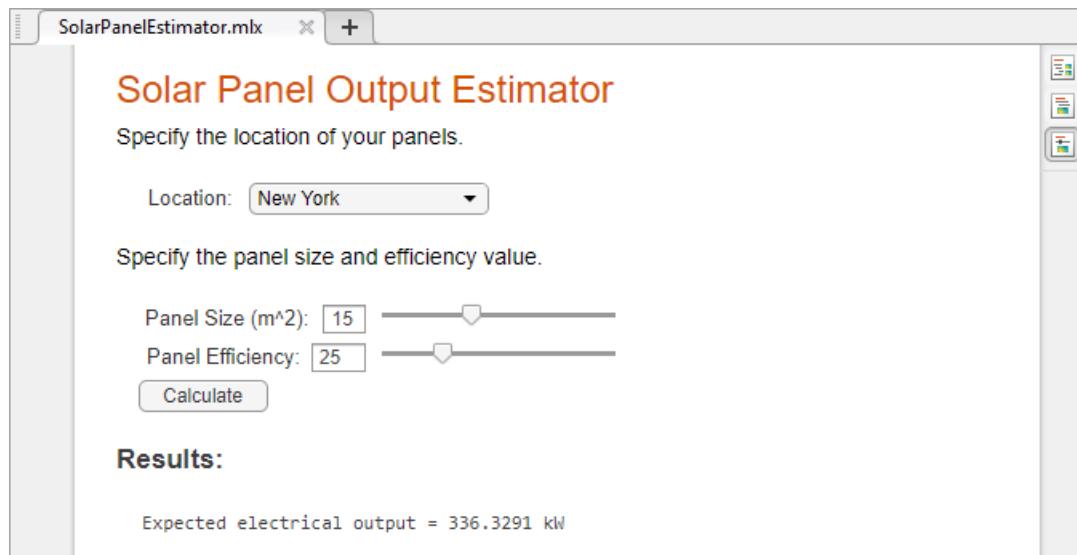
By default, output displays to the right of the code. Each output displays with the line that creates it. This option is ideal when writing code.



To display output in line with the code, select the output inline button to the right of the live script. You also can go to the **View** tab and in the **View** section, select the **Output inline** button. MATLAB displays each output underneath the line that creates it. This view is ideal for sharing.



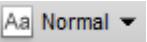
To display only output, controls, and formatted text, and hide the code, select the  hide code button to the right of the live script or in the **View** tab. This view is ideal for sharing when you want others to only change the value of the controls in your live script, or when you do not want others to see your code.



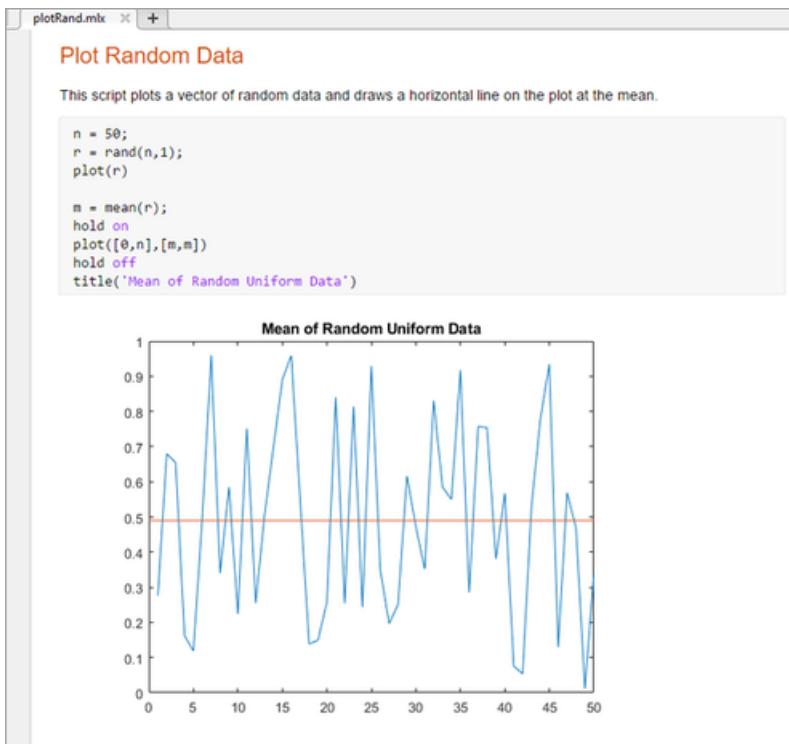
To change the default location of the output when creating a new live script, on the **Home** tab, in the **Environment** section, click  **Preferences**. Select **MATLAB > Editor / Debugger > Display**, and then select a different option for the **Live Editor default view**.

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 19-32.



To adjust the displayed font size in the Live Editor, use the **Ctrl + Plus (+)** and **Ctrl + Minus (-)** keyboard shortcuts or the **Ctrl + Mouse Scroll** keyboard shortcut. On macOS systems, use the **Command** key instead of the **Ctrl** key.

The change in the displayed font size is not honored when exporting the live script to PDF, Microsoft Word, HTML, or LaTeX.

Save Live Scripts as Plain Code

To save a live script 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 (UTF-8) (*.m)** as the **Save as type**.
- 3 Click **Save**.

When saving, MATLAB converts all formatted content to publish markup.

See Also

More About

- “Format Files in the Live Editor” on page 19-32
- “Run Sections in Live Scripts” on page 19-13
- “Modify Figures in Live Scripts” on page 19-23

- MATLAB Live Script Gallery

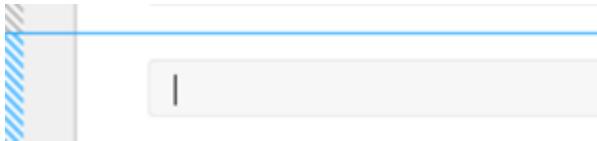
Run Sections in Live Scripts

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
Run the code in the selected section.	<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>
Run the code in the selected section, and then move to the next section.	<p>On the Live Editor tab, in the Section section, select  Run and Advance.</p>

Operation	Instructions
Run the code in the selected section, and then run all the code after the selected section.	On the Live Editor tab, in the Section section, select  Run to End .
Run all the code in the file.	On the Live Editor tab, in the Run section, click  Run .

See Also

Related Examples

- “Modify Figures in Live Scripts” on page 19-23
- “Debug Code in the Live Editor” on page 19-15

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 Run to Here button.
- Step into functions and scripts while paused by using the Step In button.
- Add breakpoints to your file to enable pausing at specific lines when you run.

In MATLAB Online, debugging in the Editor matches the Live Editor behavior. For more information about debugging in the Editor in an installed version of MATLAB, see “Debug a MATLAB Program” on page 22-2.

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 `plotRand.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.

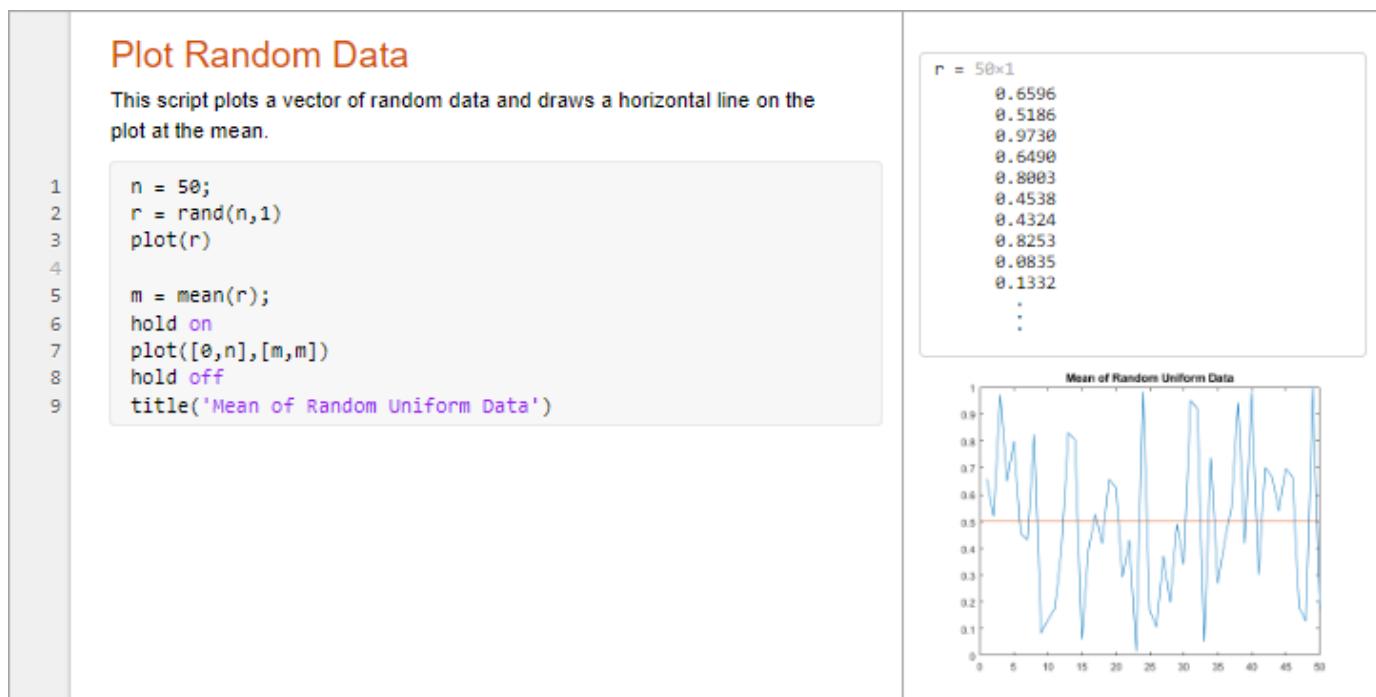
```

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')

```

The figure shows a line plot titled "Mean of Random Uniform Data". The x-axis ranges from 0 to 50 with major ticks every 5 units. The y-axis ranges from 0 to 1 with major ticks every 0.1 units. A blue line represents a vector of 50 random uniform values. A solid red horizontal line is drawn across the plot at a y-value of approximately 0.5, representing the mean of the random 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`.

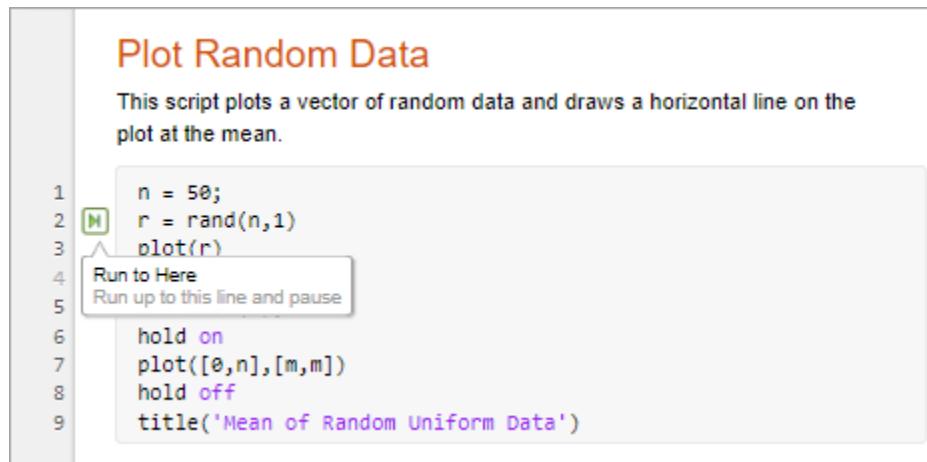


Debug Using Run to Here

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 and then pause before running the specified line of code.

To run to a specified line of code and then pause, click the run to here 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. The run to here button is only available in live functions when debugging.

For example, click the button to the left of line 2 in `plotRand.mlx`. MATLAB runs `plotRand.mlx` starting at line 1 and pauses before running line 2.



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.

```

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')

```

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.

```

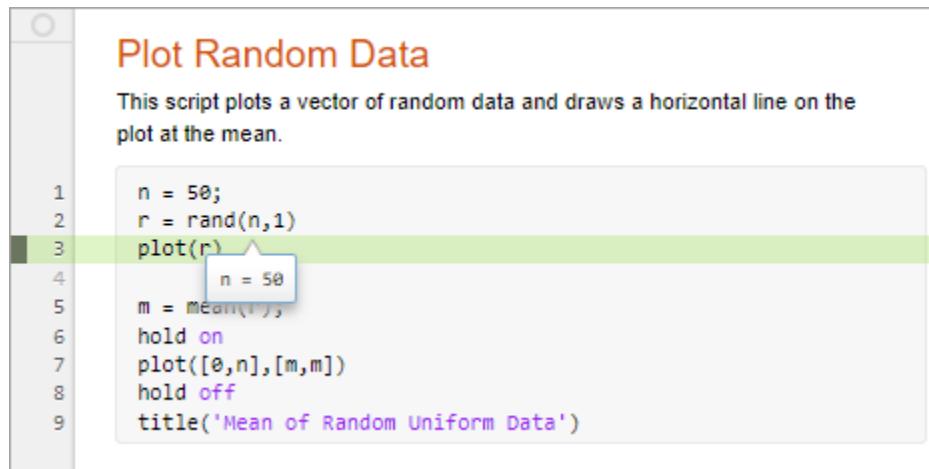
1 n = 50;
2 r = rand(n,1)
3 plot(r)
4
5 m = mean(r);
6 hold on
7 plot([0,n],[m,m])
8 hold off
9 title('Mean of Random Uniform Data')

```

You also can run up to the line with the cursor by going to the **Live Editor** tab, selecting **Step** and then selecting **Run to Cursor**.

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.



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. You also can use the **F11** key to step into a function. The Live Editor only displays the button if the line contains a call to another function.

By default, the  button only shows for user-defined functions and scripts. To show the button for 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, click the  button at the top of the file to run the rest of the called function, leave the called function, and then pause. You also can use **Shift+F11** to step out of a function.

You also can step in and out of functions by going to the **Live Editor** tab, selecting  **Step** and then selecting  **Step In** or  **Step Out**. These buttons do not honor the **Only show Step in button for user-defined functions** preference and always step in and out of both user-defined and MathWorks functions.

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. You also can use the **F12** key to set a breakpoint.

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 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. You also can use the **F12** key to clear the breakpoint.

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**. 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 `plotRand 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')
```

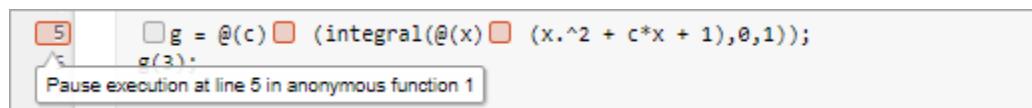
When you run the file, MATLAB pauses at the specified line when the condition is met. For example, in the `plotRand` example, MATLAB pauses before running line 3 if any of the values in `r` are equal to 0.

Add Breakpoints in Anonymous Functions

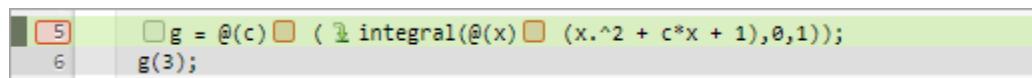
You can add multiple breakpoints in a line of MATLAB code that contains anonymous functions. You can set a breakpoint for the line itself and for each anonymous function in the line.

To add a breakpoint, click the gray area to the left on an executable line to add a breakpoint for the line. MATLAB adds a breakpoint for the line, and a disabled breakpoint for each anonymous function in the line. To enable a breakpoint for an anonymous function, right-click it and select **Enable Breakpoint**.

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.



When you set a breakpoint in an anonymous function, MATLAB pauses when the anonymous function is called. The line highlighted in green is where the code defines the anonymous function. The line highlighted in gray is where the code calls the anonymous functions. For example, in this code, MATLAB pauses the program at a breakpoint set for the anonymous function `g`, defined at line 5, and called at line 6.



See Also

More About

- “Run Sections in Live Scripts” on page 19-13

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 toolbar.

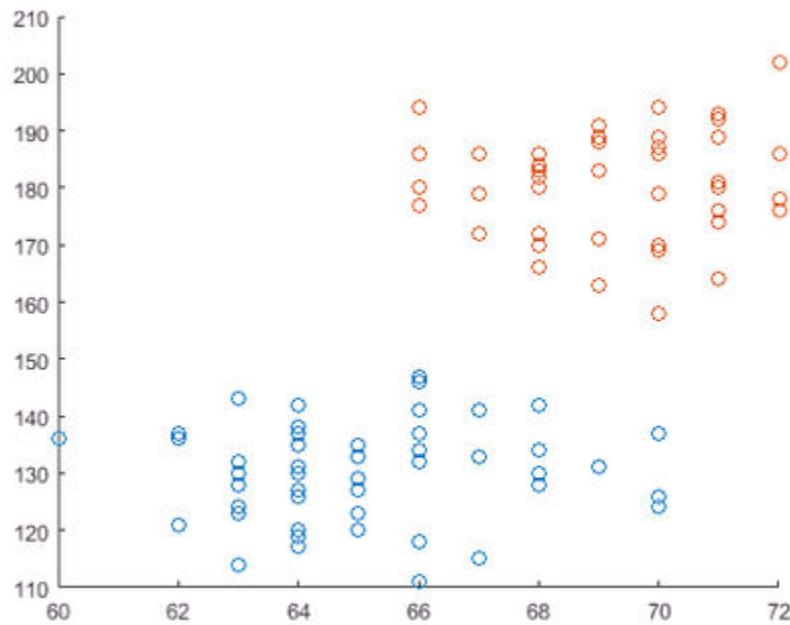
Note

- 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.
-

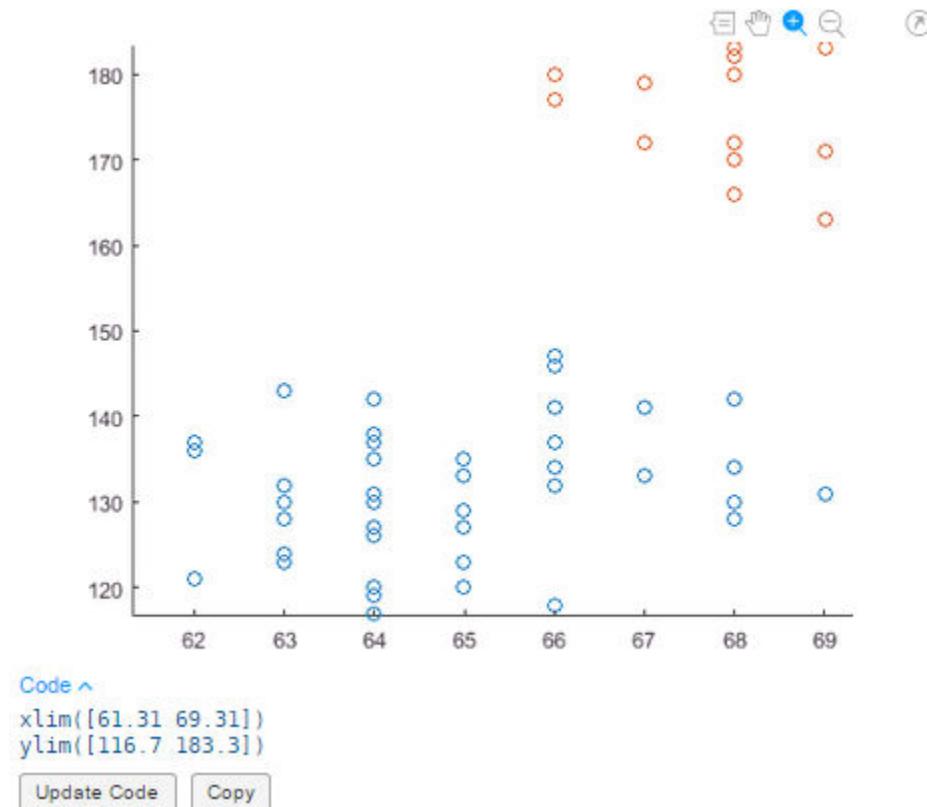
Suppose that you want to explore the health information for 100 different patients. Create a live script called `patients mlx` and add code that loads the data and adds a scatter plot that shows the height versus weight of two groups of patients, female and male. Run the code by going to the **Live Editor** tab and clicking **Run**.

```
load patients

figure
Gender = categorical(Gender);
scatter(Height(Gender=='Female'),Weight(Gender=='Female'));
hold on
scatter(Height(Gender=='Male'),Weight(Gender=='Male'));
hold off
```



Explore the points where the patient height is 64 inches. Select the button and click one of the data points where height is 64. MATLAB zooms into the 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 button. To undo or redo a formatting or annotation action, click  or  at the upper right corner of the toolstrip.

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.

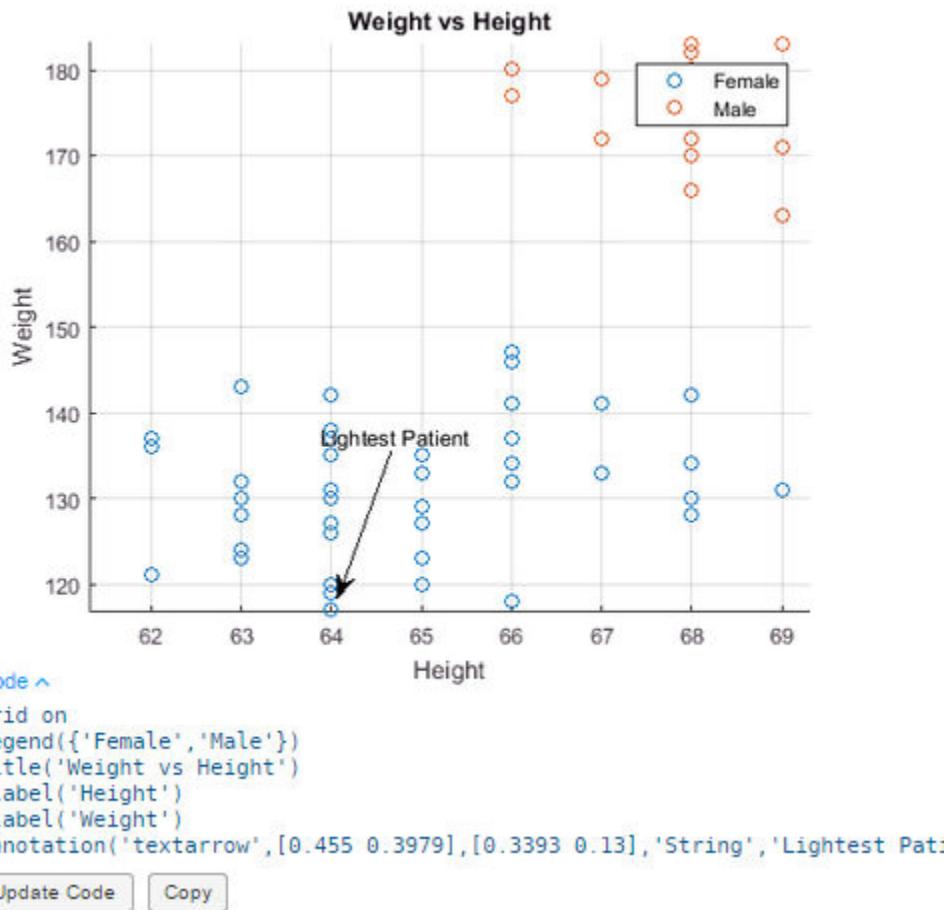
5

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');
```



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
```

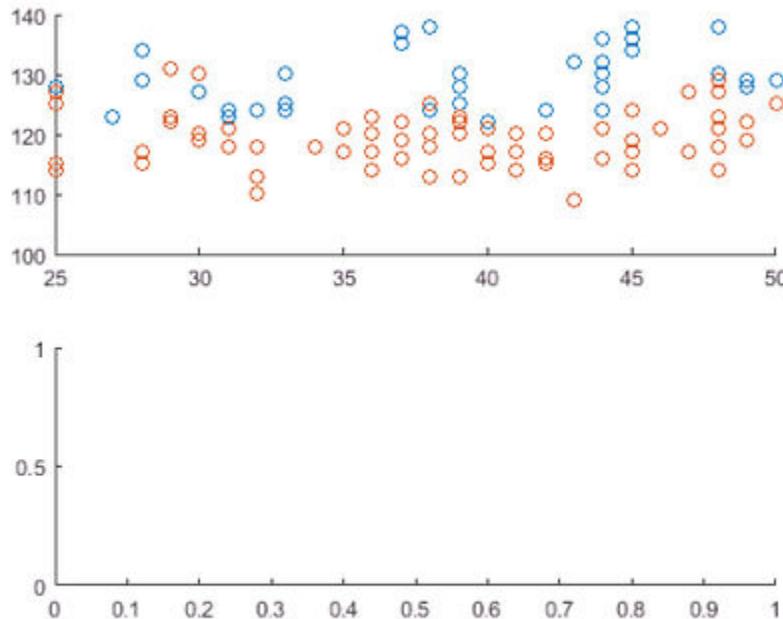
Run the code by going to the **Live Editor** tab and clicking **Run**.

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)
```

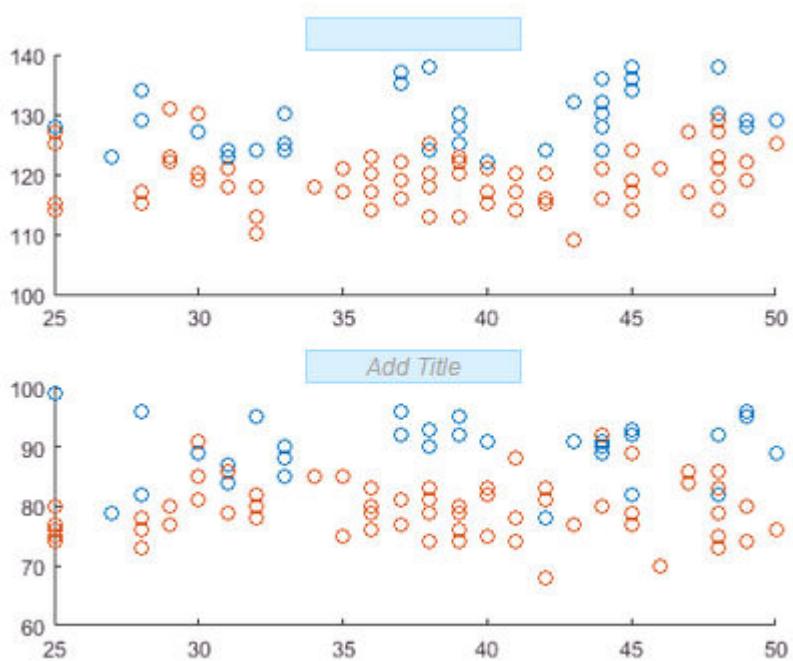
Add a scatter plot that shows the diastolic blood pressure of patients that smoke versus the diastolic blood pressure of patients that do not smoke. Run the code.

```
scatter(Age(Smoker==1),Diastolic(Smoker==1));  
hold on  
scatter(Age(Smoker==0),Diastolic(Smoker==0));  
hold off
```

Add formatting:

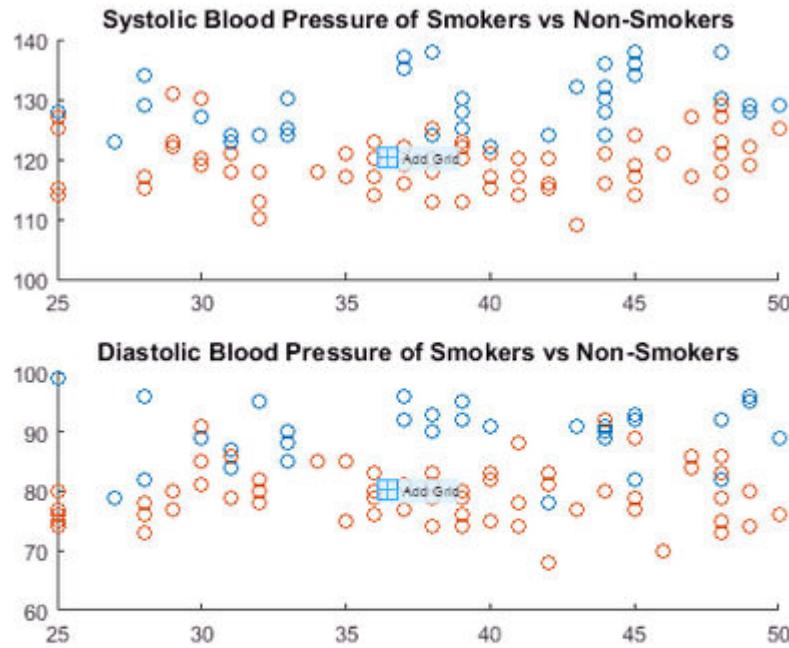
1

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** button appears on each subplot. Click the **Add Grid** button on each subplot. Grid lines appear in both subplots.

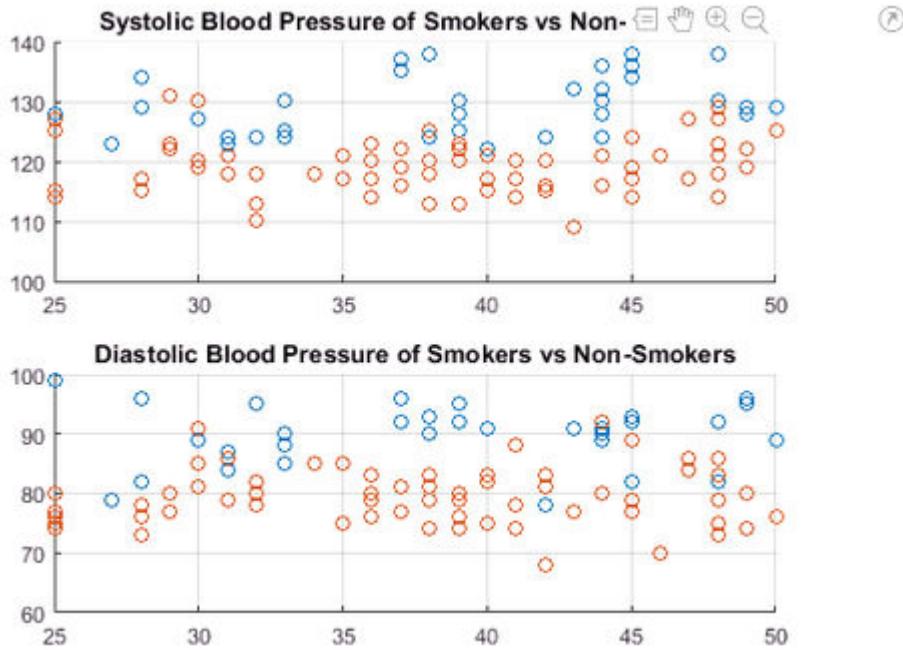


Code ^
subplot(2,1,1)
title('Systolic Blood Pressure of Smokers vs Non-Smokers')
subplot(2,1,2)
title('Diastolic Blood Pressure of Smokers vs Non-Smokers')

[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  button 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 Plot as Image or Vector Graphics File” 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 19-32
- “Run Sections in Live Scripts” on page 19-13

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 19-13.
 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 insert 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. <ul style="list-style-type: none">• Select Plain to insert sample code as unhighlighted text.• Select MATLAB to insert sample code as text highlighted according to MATLAB syntax.

Option	Description	Additional Details
 Image	Insert an image.	<p>You can insert 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.</p> <p>To change the alternate text, alignment, and size of an image after inserting it, right-click the image and select Edit Image... from the context menu.</p> <ul style="list-style-type: none"> • Alt Text — Add text to the edit field to specify alternative text for the image. • Alignment — Select from the available options to specify how the image aligns with the other items in the row. • Size — To specify a size relative to the original image size, select Relative (%) and specify the width and height of the image as a percentage of the original image. To specify an absolute size, select Absolute (px) and specify the width and height of the image in pixels. Select Keep Aspect Ratio to maintain the aspect ratio while resizing. <p>To return to the original image size, right-click the image and select Reset Image.</p>
 Hyperlink	Insert a hyperlink.	<p>You can insert 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> <ul style="list-style-type: none"> • Select Web page to insert a hyperlink to an external web page. Then, enter the URL of the web page. • Select Location in document 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. You also can use the Alt + Up Arrow and Alt + Down Arrow keyboard shortcuts. Location can be a code section, text paragraph, title, or heading. Linking to individual lines of text or code is not supported. • Select Existing file to insert a hyperlink to a file. Then, enter the file path.
 Equation	Insert an equation.	<p>You can insert 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 19-38.</p>

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
Text Alignment	Left Center Right
Lists	Numbered list Bulleted list
Standard Formatting	Bold Italic Underline 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.

Change Fonts

You can adjust the displayed font size in the Live Editor, or use settings to change the font name, style, size, and color of code and text.

To increase or decrease the displayed font size in the Live Editor, zoom in or out using the **Ctrl + Plus (+)** and **Ctrl + Minus (-)** keyboard shortcuts or by holding **Ctrl** and scrolling with the mouse. On macOS systems, use the **Command** key instead of the **Ctrl** key. The change in the displayed font size is not honored when exporting the live script to PDF, Microsoft Word, HTML, or LaTeX.

To change the font name, style, size, and color of code and text, using settings. For example, this code changes the color and style of titles in the Live Editor:

```
s = settings;
s.matlab.fonts.editor.title.Style.PersonalValue = {'bold'};
s.matlab.fonts.editor.title.Color.PersonalValue = [0 0 255 1];
```

This code increases the size and changes the font name of normal text in the Live Editor:

```
s = settings;
s.matlab.fonts.editor.normal.Size.PersonalValue = 20;
s.matlab.fonts.editor.normal.Name.PersonalValue = 'Calibri';
```

The Live Editor updates all open live scripts and live functions to show the selected fonts. When you create new live scripts or functions, the selected fonts are applied as well.

Plot Random Data

This script 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')
```

For more information, see `matlab.fonts`

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	# text + Enter	Ctrl + Alt + L
Heading 1	## text + Enter	Ctrl + Shift + 1
Heading 2	### text + Enter	Ctrl + Shift + 2
Heading 3	#### text + Enter	Ctrl + Shift + 3

Formatting Style	Autoformatting Sequence	Keyboard Shortcut
Section break with heading 1	<code>%% text + Enter</code>	With cursor at beginning of line with <i>text</i> : Ctrl + Shift + 1 , then Ctrl + Alt + Enter
Section break	<code>%% + Enter</code> <code>- - - + Enter</code> <code>*** + Enter</code>	Ctrl + Alt + Enter
Bulleted list	<code>* text</code> <code>- text</code> <code>+ text</code>	Ctrl + Alt + U
Numbered list	<code>number. text</code>	Ctrl + Alt + O
Italic	<code>*text*</code> <code>_text_</code>	Ctrl + I
Bold	<code>**text**</code> <code>__text__</code>	Ctrl + B
Bold and italic	<code>***text***</code> <code>___text___</code>	Ctrl + B , then Ctrl + I
Monospace	<code>`text`</code> <code> text </code>	Ctrl + M
Underline	None	Ctrl + U
LaTeX equation	<code>\$LaTeX\$</code>	Ctrl + Shift + L
Hyperlink	<code>URL + Space or Enter</code> <code><URL></code> <code>[Label] (URL)</code>	Ctrl + K
Trademark, service mark, and copyright symbols (™, ®, 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”.

See Also

More About

- “Insert Equations into the Live Editor” on page 19-38
- “Share Live Scripts and Functions” on page 19-66

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 equitorial 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

$$\delta = \sin^{-1} (\sin(23.45) \sin(\frac{360}{365}(d - 81)))$$

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 = %.2f\nSolar Elevation = %.2f\n', delta, alpha)
```

There are two ways to insert an equation into a live script or function.

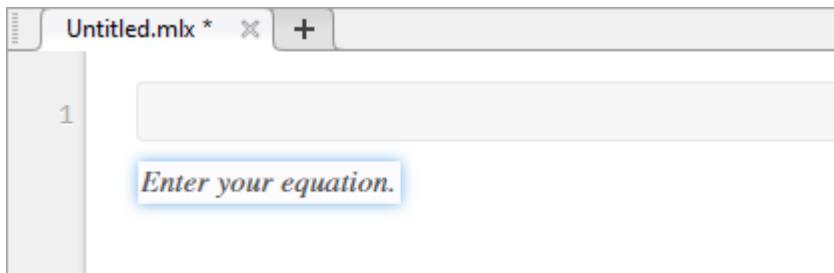
- Insert an equation interactively — You can build an equation interactively by selecting from a graphical display of symbols and structures.
- Insert a LaTeX equation — You can enter LaTeX commands and the Live Editor inserts the corresponding equation.

Insert Equation Interactively

To insert an equation interactively:

- 1 Go to the **Insert** tab and click **Σ 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 \downarrow to the right of each section.

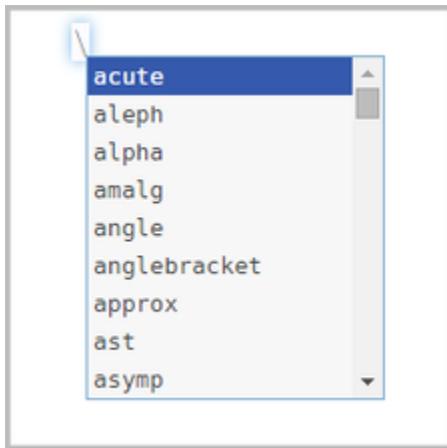
When adding or editing a matrix, a context menu appears, which you can use to delete and insert rows and columns. You also can use the context menu to change or remove matrix delimiters.

- 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 button 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>	\models	<code><--></code>	\leftrightarrow	<code>!<</code>	\prec
<code> -</code>	\vdash	<code><-></code>	\Leftrightarrow	<code>!></code>	\succ
<code>- </code>	\dashv	<code><=</code>	\leq	<code>!<=</code>	$\not\leq$
<code>-></code>	\rightarrow	<code>>=</code>	\geq	<code>!>=</code>	$\not\geq$

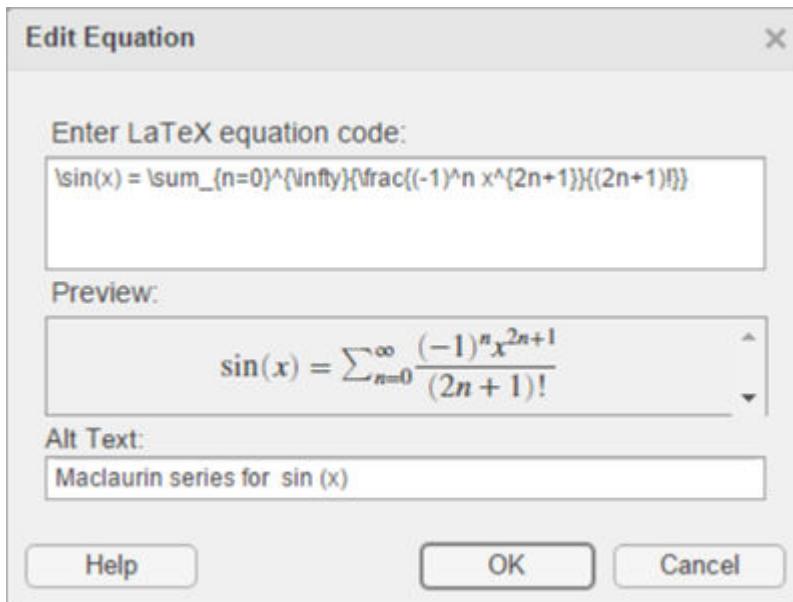
Keyboard Input	Symbol	Keyboard Input	Symbol	Keyboard Input	Symbol
<-	\leftarrow	<>	\neq		
<--	$\leftarrow\!\!\leftarrow$	$\sim=$	\neq		

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 To include a description of the LaTeX equation when exporting the live script to HTML, add text to the **Alt Text** field. For example, you can enter the text **Maclaurin series for sin(x)**.

The description specifies alternative text for the equation and is saved as an `alt` attribute in the HTML document. It is used to provide additional information for the equation if, for example, a user is using a screen reader.

- 4 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
$a^2 + b^2 = c^2$	$a^2 + b^2 = c^2$
$\int_0^2 x^2 \sin(x) dx$	$\int_0^2 x^2 \sin(x) dx$

LaTeX Expression	Equation in Live Script
$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$	$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$
$\{a,b,c\} \neq \{a,b,c\}$	$a, b, c \neq \{a, b, c\}$
$x^2 \geq 0 \quad \text{for all } x \in \mathbf{R}$	$x^2 \geq 0 \quad \text{for all } x \in \mathbf{R}$
$\begin{matrix} a & b \\ \hline c & d \end{matrix}$	$\begin{matrix} a & b \\ c & d \end{matrix}$

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	LaTeX Command
α	alpha	ν	nu	ξ	xi
β	beta	ω	omega	ζ	zeta
χ	chi	\circ	omicron	ε	varepsilon
δ	delta	ϕ	phi	φ	varphi
ε	epsilon	π	pi	ϖ	varpi
η	eta	ψ	psi	ϱ	varrho
γ	gamma	ρ	rho	ς	varsigma
ι	iota	σ	sigma	ϑ	vartheta
κ	kappa	τ	tau	\aleph	aleph
λ	lambda	θ	theta		
μ	mu	υ	upsilon		
Δ	Delta	Φ	Phi	Θ	Theta
Γ	Gamma	Π	Pi	Υ	Upsilon
Λ	Lambda	Ψ	Psi	Ξ	Xi
Ω	Omega	Σ	Sigma		

Operator Symbols

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol	LaTeX Command
*	ast	\pm	pm	\cap	cap
\star	star	\mp	mp	\cup	cup
.	cdot	\amalg	amalg	\uplus	uplus
\circ	circ	\odot	odot	\sqcap	sqc
\bullet	bullet	\ominus	ominus	\sqcup	sqc

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol	LaTeX Command
\diamond	diamond	\oplus	oplus	\wedge	wedge
\setminus	setminus	\oslash	oslash	\vee	vee
\times	times	\otimes	otimes	\triangleleft	triangleleft
\div	div	\dagger	dagger	\triangleright	triangleright
\perp	bot, perp	\ddagger	ddagger	\triangle	bigtriangleup
\top	top	\wr	wr	∇	bigtriangledown
Σ	sum	\prod	prod	\int	int
\uplus	biguplus	\oplus	bigoplus	\vee	bigvee
\cap	bigcap	\otimes	bigotimes	\wedge	bigwedge
\cup	bigcup	\odot	bigodot	\sqcup	biguplus

Relation Symbols

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol	LaTeX Command
\equiv	equiv	$<$	lt	$>$	gt
\cong	cong	\leq	le, leq	\geq	ge, geq
\neq	neq, ne, not=	\prec	not<	\succ	not>
\sim	sim	\preccurlyeq	prec	\succcurlyeq	succ
\approx	simeq	\ll	preceq	\gg	succq
\approx	approx	\lll	ll	\ggg	gg
\asymp	asymptotic	\subset	subset	\supset	supset
\doteq	doteq	\subseteq	subseteq	\supseteq	supseteq
\propto	propto	\sqsubset	sqsubset	\sqsupseteq	sqsubseteq
\models	models	$ $	mid	ϵ	in
\bowtie	bowtie	\parallel	parallel	\notin	notin
\vdash	vdash	\Leftrightarrow	iff	\ni	ni
\dashv	dashv				

Note The `leq`, `geq`, `equiv`, `approx`, `cong`, `sim`, `simeq`, `models`, `ni`, `succ`, `succq`, `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 $\not\leq$.

Arrows

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol	LaTeX Command
\leftarrow	leftarrow	\rightarrow	rightarrow	\uparrow	uparrow
\Leftarrow	Leftarrow	\Rightarrow	Rightarrow	\Updownarrow	Updownarrow
\longleftarrow	longleftarrow	\longrightarrow	longrightarrow	\downarrow	downarrow
\Longleftarrow	Longleftarrow	\Longrightarrow	Longrightarrow	\Downarrow	Downarrow

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol	LaTeX Command
\hookleftarrow	hookleftarrow	\hookrightarrow	hookrightarrow	\uparrow	update
\leftrightharpoonup	leftharpoondown	\rightharpoonup	rightharpoondown	\Downarrow	Update
\leftharpoonup	leftharpoonup	\rightharpoonup	rightharpoonup	\leftrightarrow	left
\swarrow	swarrow	\nearrow	nearrow	\Leftrightarrow	Left
\nwarrow	narrow	\searrow	searrow	\longleftrightarrow	long row
\mapsto	mapsto	\longmapsto	longmapsto	\Longleftrightarrow	Long row

Brackets

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol	LaTeX Command
{	lbrace	}	rbrace		vertical
[lbrack]	rbrack		Vertical
<	langle	>	rangle	\backslash	backward
[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	LaTeX Command
∞	infty	\forall	forall	\wp	wp
∇	nabla	\exists	exists	\angle	ang
∂	partial	\emptyset	emptyset	\triangle	tri
\Im	Im	\mathbf{i}	i	\hbar	hbar
\Re	Re	\mathbf{j}	j	$'$	prime
ℓ	ell	\mathbf{l}	imath	\neg	lneg
...	dots, ldots	\mathbf{J}	jmath	\vee	sur

Symbol	LaTeX Command	Symbol	LaTeX Command	Symbol	LaTeX Command
...	cdots	:	colon	\leftarrow	get
\ddots	ddots	.	cdotp	\rightarrow	to
\vdots	vdots	.	ldotp		

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	LaTeX Command
\acute{a}	acute	\ddot{a}	ddot	\tilde{a}	tilde
\bar{a}	bar	\dot{a}	dot	\vec{a}	vec
\breve{a}	breve	\grave{a}	grave		
\check{a}	check	\widehat{a}	hat		

Functions

Sample	LaTeX Command	Sample	LaTeX Command	Sample	LaTeX Command
\arccos	<code>arccos</code>	\det	<code>det</code>	\ln	<code>ln</code>
\arcsin	<code>arcsin</code>	\dim	<code>dim</code>	\log	<code>log</code>
\arctan	<code>arctan</code>	\exp	<code>exp</code>	\max	<code>max</code>
\arg	<code>arg</code>	\gcd	<code>gcd</code>	\min	<code>min</code>
\cos	<code>cos</code>	\hom	<code>hom</code>	\Pr	<code>Pr</code>
\cosh	<code>cosh</code>	\ker	<code>ker</code>	\sec	<code>sec</code>
\cot	<code>cot</code>	\lg	<code>lg</code>	\sin	<code>sin</code>
\coth	<code>coth</code>	\lim	<code>lim</code>	\sinh	<code>sinh</code>
\csc	<code>csc</code>	\liminf	<code>liminf</code>	\sup	<code>sup</code>
\deg	<code>deg</code>	\limsup	<code>limsup</code>	\tan	<code>tan</code>

Math Constructs

Sample	LaTeX Command	Sample	LaTeX Command	Sample	LaTeX Command
$\frac{abc}{xyz}$	<code>frac</code>	$\frac{a}{b}$	<code>over</code>	$\frac{a}{b}$	<code>stacove</code>
\sqrt{abc}	<code>sqrt</code>	$\left[\frac{a}{b} \right]$	<code>overwithdelims</code>	$\frac{b}{a}$	<code>und</code>
$\mod a$	<code>bmod</code>	\overleftarrow{abc}	<code>overleftarrow</code>	$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$	<code>pma</code>
$(\mod a)$	<code>pmod</code>	\overrightarrow{abc}	<code>overrightarrow</code>	$a \ b$ $c \ d$	<code>mat</code>

Sample	LaTeX Command	Sample	LaTeX Command	Sample	LaTeX Command
\widehat{abc}	widehat	\overleftarrow{abc}	overleftrightarrow row	$a b$ $c d$	beg
\widetilde{abc}	widetilde	\int_a^b	limits	$\begin{cases} a \\ c \end{cases} \begin{cases} b \\ d \end{cases}$	beg
	left		right	\overline{ab} \underline{cd}	hli

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	LaTeX Command
ab	negthinspace	abc	mathord	$a[b$	mat
ab	thinspace	$a\sum b$	mathop	$a]b$	mat
$a b$	enspace	$a + b$	mathbin	$a b$	mat
$a~b$	quad	$a = b$	mathrel	$a~b$	ker
$a~~b$	qquad	a,b	mathpunct		

Text Styling

Sample	LaTeX Command	Sample	LaTeX Command	Sample	LaTeX Command
Σ	displaystyle	ABCDE	text, textnormal	$ABCDE$	tex
Σ	textstyle	ABCDE	bf, textbf, mathbf	$ABCDE$	tex
Σ	scriptstyle	<i>ABCDE</i>	it, textit, mathit	$A\mathcal{B}CD\mathcal{E}$	cal
Σ	scriptscriptstyle	ABCDE	rm, textrm, mathrm	ABCDE	hbo

See Also

Related Examples

- “Share Live Scripts and Functions” on page 19-66

External Websites

- <https://www.latex-project.org/>

Add Interactive Controls to a Live Script

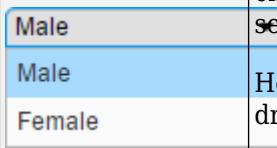
You can add sliders, drop-downs, check boxes, edit fields, and buttons to your live scripts to control variable values interactively. Adding interactive controls to a script is useful when you want to share the script with others. Use interactive controls to set and change the values of variables in your live script using familiar user interface components.

Insert Controls

To insert a control into a live script, go to the **Live Editor** tab, and in the **Code** section, click **Control**. Then, select from the available options. To replace an existing value with a control, select the value and then insert the control. The **Control** menu only shows options available for the selected value.

The table below shows the list of available controls:

Control	Description	Configuration Details
Numeric Slider 	<p>Use numeric sliders to interactively change the value of a variable by moving the slider thumb to the desired numeric value.</p> <p>The value to the left of the slider is its current value.</p>	<p>In the Values section, specify a Min, Max, and Step value or select a workspace variable from the drop-down list.</p> <p>Only variables with numeric values appear in the drop-down list. If the variables that you want to select are not listed, try running the live script first to create the variables and add them to the workspace. After running the live script, changes to the variable are automatically reflected in the numeric slider.</p>

Control	Description	Configuration Details
Drop-Down List	<p>Use drop-down lists to interactively change the value of a variable by selecting from a list of values.</p>  <p>Male</p> <p>Male</p> <p>Female</p>	<p>In the Items > Item Labels field, specify the text that you want to display for each item in the drop-down list.</p> <p>In the Items > Item Values field, specify the values for each item in the drop-down list. Make sure to enclose text values in quotes or double quotes, because the Live Editor interprets each item in the list as code.</p> <p>To populate the items in the drop-down list using values stored in a variable, in the Items > Variable field, select a workspace variable. The variable must be a string array to appear in the list. If the variable that you want to select is not listed, try running the live script first to create the variable and add it to the workspace. After running the live script, changes to the variable are automatically reflected in the drop-down list.</p>
Check Box	<p>Use check boxes to interactively set the value of a variable to either the logical value 1 (true) or the logical value 0 (false).</p>  <p>The displayed state of the check box (checked or not checked) determines its current value.</p>	N/A
Edit Field	<p>Use edit fields to interactively set the value of a variable to any typed input.</p>  <p>The text displayed in the edit field and the selected data type determines its current value.</p>	<p>In the Type section, in the Data type field, select from the available options to specify the data type of the text in the edit field.</p>
Button	<p>Use button controls to interactively run code on button click.</p>  <p>When using button controls, consider setting the Run field for all other controls in the live script to None. Then, the code only runs when the user clicks the button control. This can be useful when the live script requires multiple control values to be set before running the code.</p>	<p>To change the label displayed on the button, in the Label section, enter a label name.</p>

Modify Control Execution

You can modify when and what code runs when the value of a control changes. By default, when the value of a control changes, the Live Editor runs the code in the current section. To configure this behavior, right-click the control and select **Configure Control**. Then, in the **Execution** section, modify the values of the fields described in the table below. Press **Tab** or **Enter**, or click outside of the control configuration menu to return to the live script.

Field	Options
Run On (slider control only)	Select one of these options to specify when the code runs: <ul style="list-style-type: none"> • Value changing — Run the code while the value of the slider is changing. • Value changed — Run the code after the slider value is done changing (user has released the slider thumb).
Run	Select one of these options to specify what code runs when the value of the control changes: <ul style="list-style-type: none"> • Current section (Default) — Run the section that includes the control. • Current section and modified or not yet run sections above — Run the current section and any stale code above it when the control value changes. If the live script has not yet been run, changing the control value will run the current section and all sections before it. • Current section to end — Run the section that includes the control and any sections that follow. • All Sections — Run all sections in the live script. • Nothing — Do not run any code. <p>Tip When using a button control in a live script, consider setting the Run field for all other controls in the live script to Nothing. Then, the code only runs when the user clicks the button control. This can be useful when the live script requires multiple control values to be set before running the code.</p>

Modify Control Labels

You can hide the code in a live script and only display labeled controls, output, and formatted text. Hiding the code is useful when sharing and exporting live scripts. To hide the code, click the  hide code button to the right of the live script. You also can go to the **View** tab, and in the **View** section,

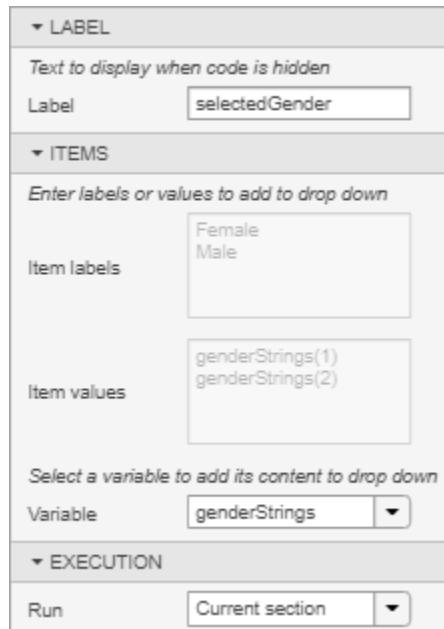
click **Hide Code**. To show the code again, click the output inline button or the output on right button.

When the code is hidden, labels display next to the control. To modify the label for a control, right-click the control and select **Configure Control**. Then, in the **Label** section, enter a label name. This is also the text that displays on button controls in all views. Press **Tab** or **Enter**, or click outside of the control configuration menu to return to the live script.

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.

To specify the gender of the patients to plot, insert a drop-down list and select the `genderStrings` variable to populate the items in the list. To specify a threshold height and weight, insert two numeric sliders and select the `minHeight`, `maxHeight`, `minWeight`, and `maxWeight` variables as the **Min** and **Max** values.



To view and interact with the controls, open this example in your browser or in MATLAB.

```
load patients
genderStrings = ["Female", "Male"];

selectedGender =  ;
minHeight = min(Height);
maxHeight = max(Height);
minWeight = min(Weight);
maxWeight = max(Weight);

thresholdHeight =  ;
```

```

thresholdWeight = 132  ;
```

```

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
```

```

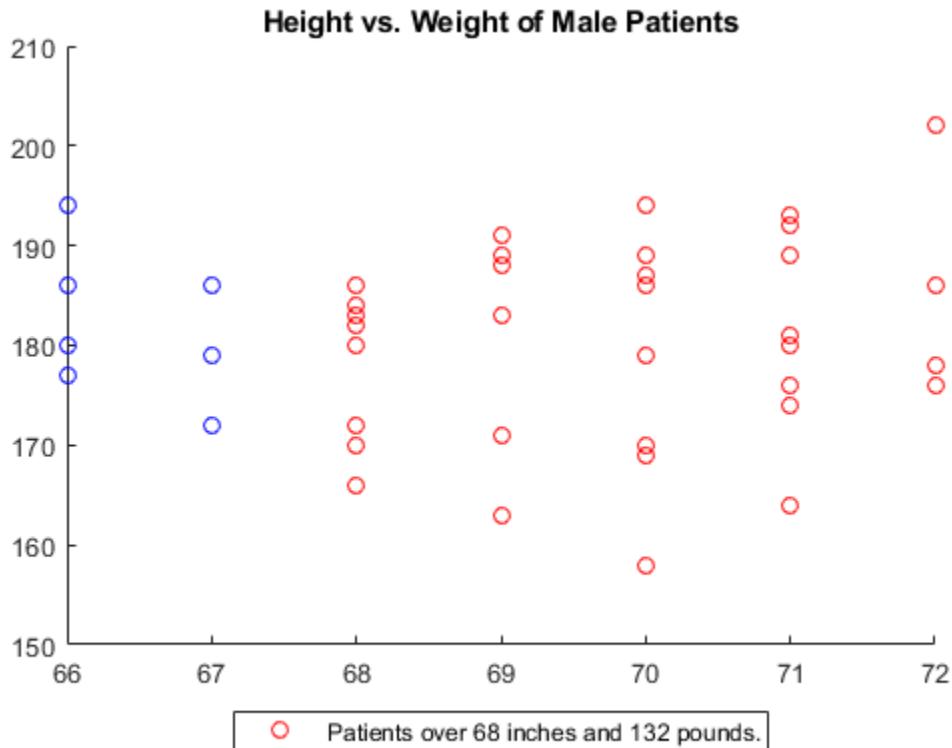
sp2 = scatter(overThresholdHeights, overThresholdWeights, 'red');
hold off
```

```

title('Height vs. Weight of ' + selectedGender + ' Patients')
```

```

legendText = sprintf('Patients over %d inches and %d pounds.',thresholdHeight,thresholdWeight);
legend(sp2,legendText,'Location','southoutside')
```



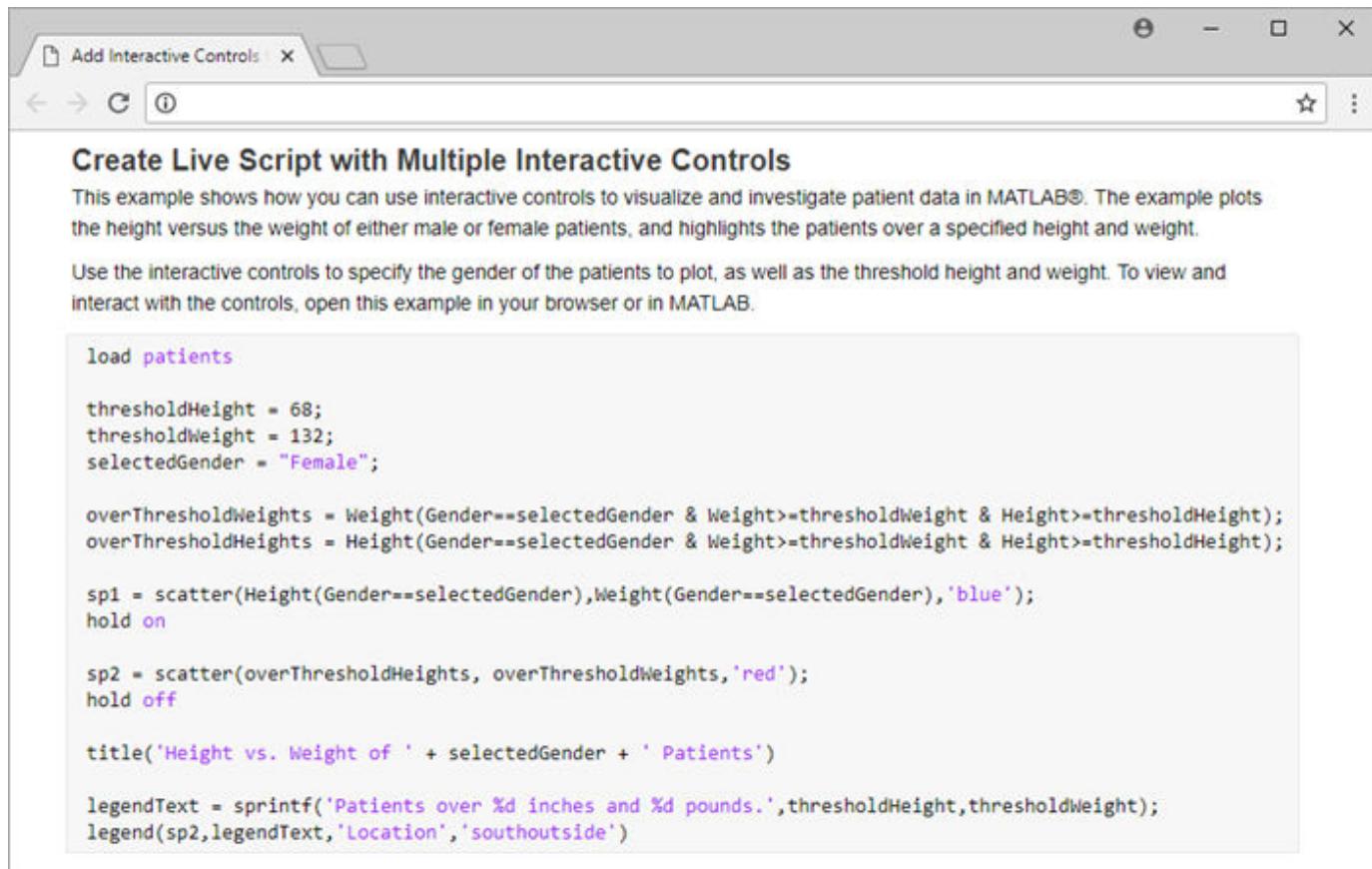
Share Live Script

When the live script is complete, share it with others. Users can open the live script in MATLAB and experiment with using the controls interactively.

If you share the live script itself as an interactive document, consider hiding the code in the live script before sharing it. When the code is hidden, the Live Editor only displays labeled controls,

output, and formatted text. To hide the code, click the  hide code button to the right of the live script. You also can go to the **View** tab, and in the **View** section, click  **Hide Code**.

If you share the live script as a static PDF, Microsoft Word, 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").



```
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

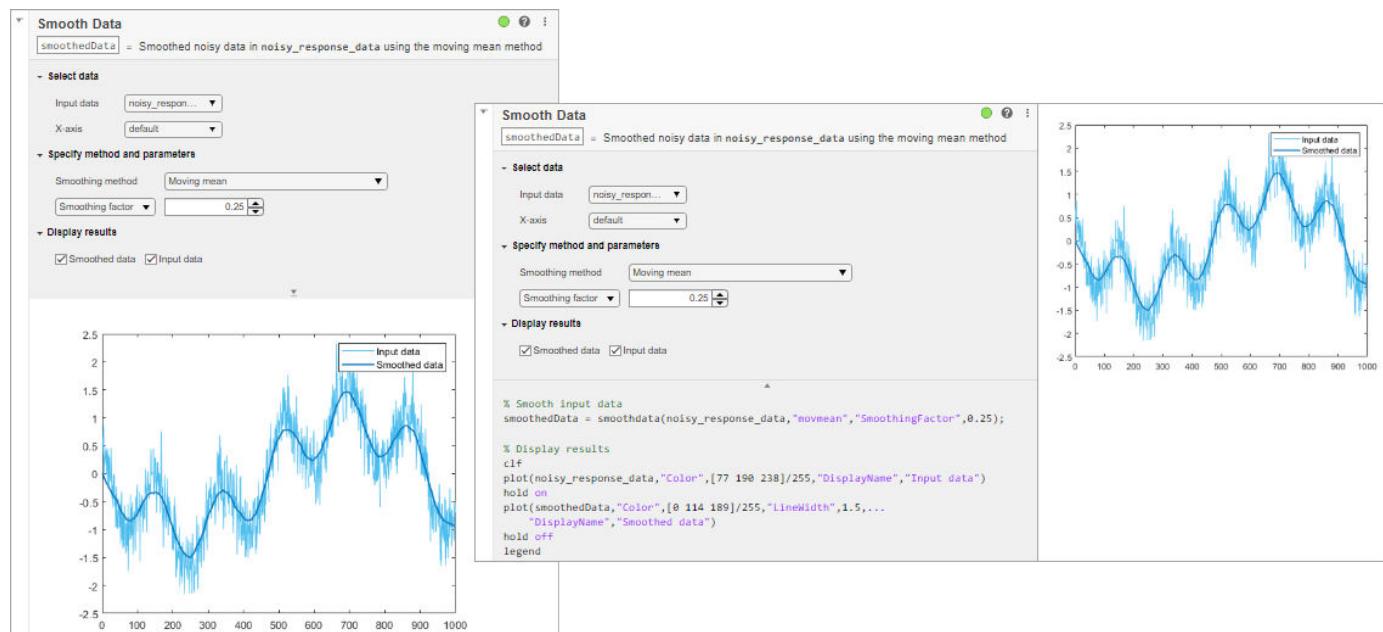
- “Add Interactive Tasks to a Live Script” on page 19-53
- “Share Live Scripts and Functions” on page 19-66

Add Interactive Tasks to a Live Script

What Are Live Editor Tasks?

Live Editor tasks are apps that can be added to a live script to perform a specific set of operations. You can add tasks to live scripts to explore parameters and automatically generate code. Use tasks to reduce development time, errors, and time spent plotting.

Tasks represent a series of MATLAB commands. You can display their output either inline or on the right. To see the MATLAB commands that the task runs, show the generated code.

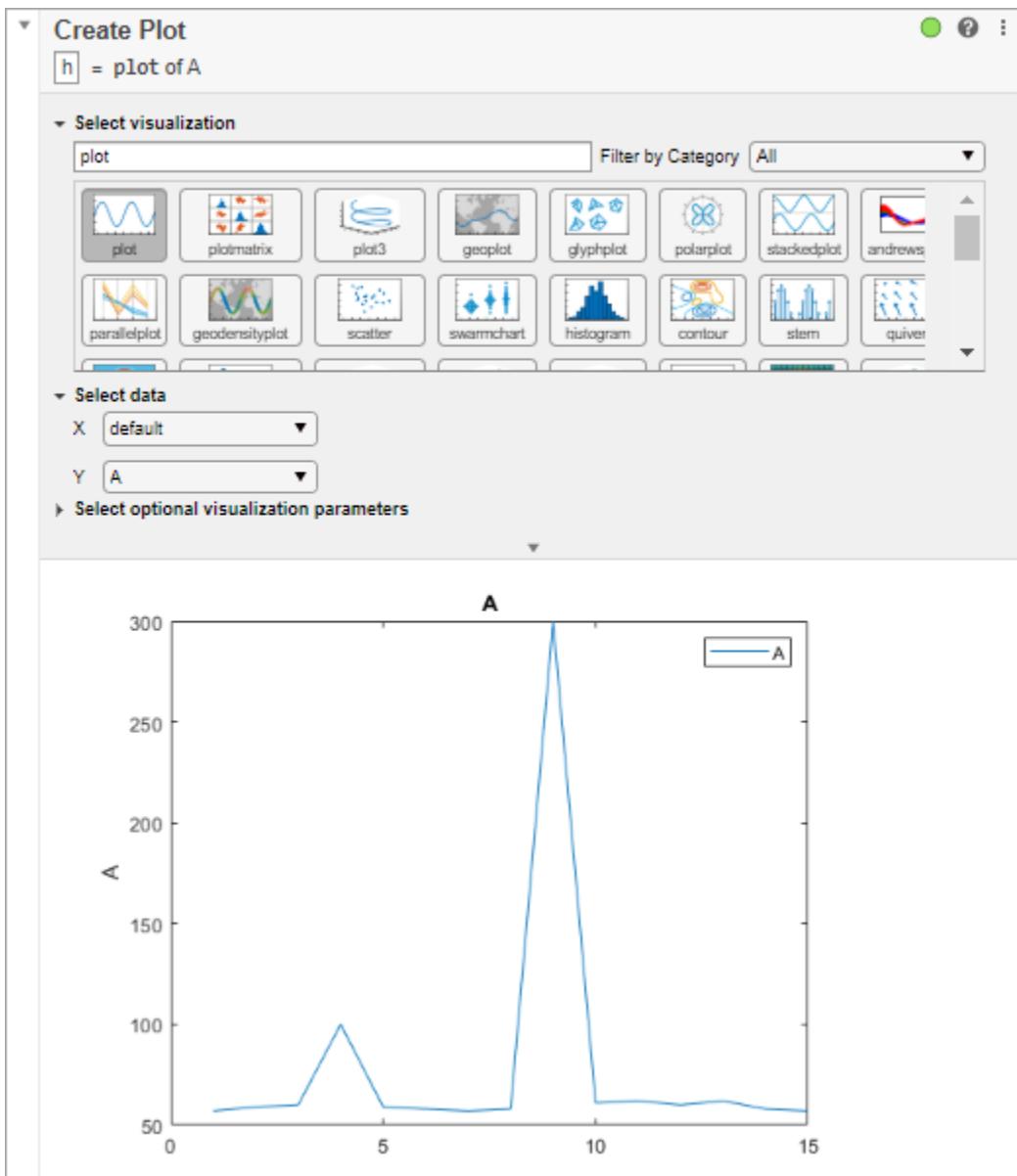


Insert Tasks

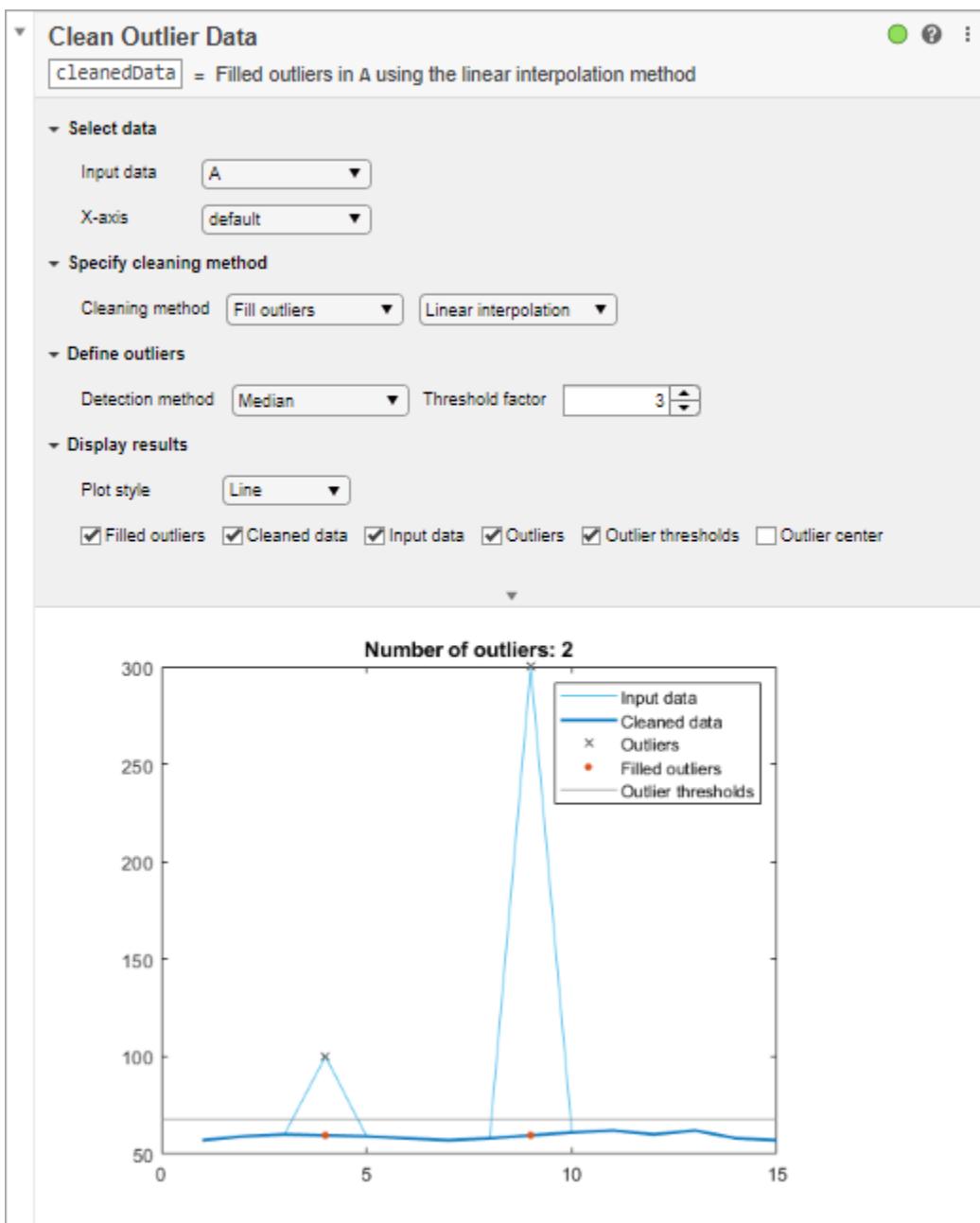
To add a task to a live script, go to the **Live Editor** tab, click **Task** **▼**, and select from the available tasks. You also can type the name of the task in a live script code block. As you type, the Live Editor displays possible matches, and you can select and insert the desired task. For example, create a live script that creates a vector of data containing an outlier.

```
A = [57 59 60 100 59 58 57 58 300 61 62 60 62 58 57];
```

Add the **Create Plot** task to your live script to plot the vector of data.



Add the **Clean Outlier Data** task to your live script to smooth the noisy data and avoid skewed results. To add the task, start typing the word `clean` in the live script and select `Clean Outlier Data` from the suggested command completions. In the task, set **Input data** to `A`. The task identifies and fills two outliers in the data and creates the variable `cleanedData` in the MATLAB workspace with the stored results. You also can see the results in the output plot for the task. Continue modifying additional parameters until you are satisfied with the results.

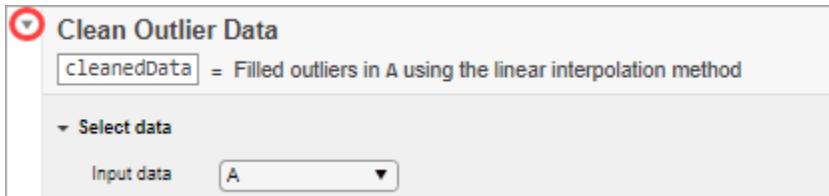


Restore Default Parameters

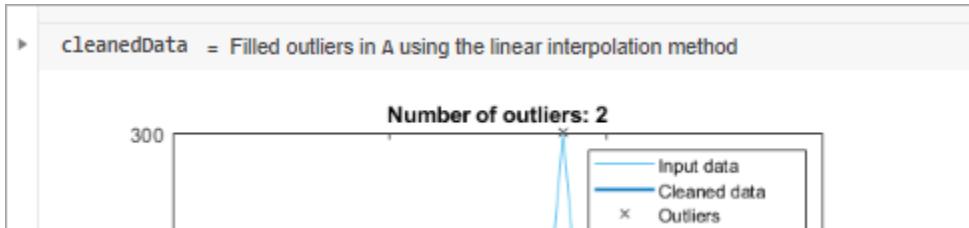
To restore all parameter values back to their defaults, click the options button : at the top-right of the task and select **Restore Default Values**.

Collapse Tasks for Improved Readability

When you are done modifying parameters, you can collapse the task to help with readability. To collapse the task, click the arrow at the top-left of the task.



The task displays as a single, user-readable line of pseudocode with output.



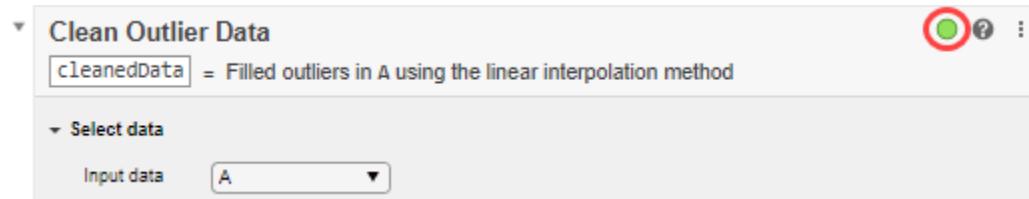
Delete Tasks

To delete a task, click the task and then press **Delete** or **Backspace**. You also can place your cursor directly before or after the task and use the **Delete** or **Backspace** key, respectively.

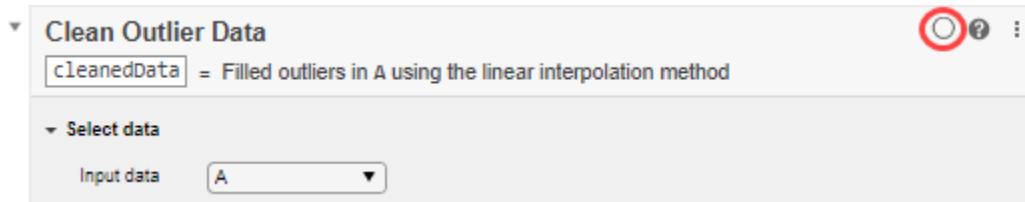
Run Tasks and Surrounding Code

By default, as you modify the value of parameters in the task, the task and current section (including other tasks in the section) run automatically. This ensures that the results and surrounding code in the section remain up to date. For example, in the live script `cleanmydata mlx`, the entire section including the code that creates the vector of noisy data reruns every time you modify the value of a parameter in the Clean Outlier Data task. To run only the task, add section breaks before and after the task. For more information about sections and how to add section breaks, see “Run Sections in Live Scripts” on page 19-13.

A green circular icon at the top-right of the task indicates that the task runs automatically when you modify the task parameters.



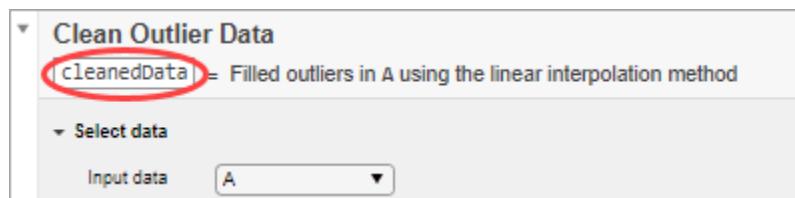
To disable running the section automatically, click the autorun icon. The icon updates to display the disabled state. To run the task and current section, on the **Live Editor** tab, click the **Run Section** button.



Some tasks do not run automatically by default. This default setting ensures optimal performance for those tasks.

Modify Output Argument Name

To modify the name of the output argument, click the text box containing the argument name and enter a new name.

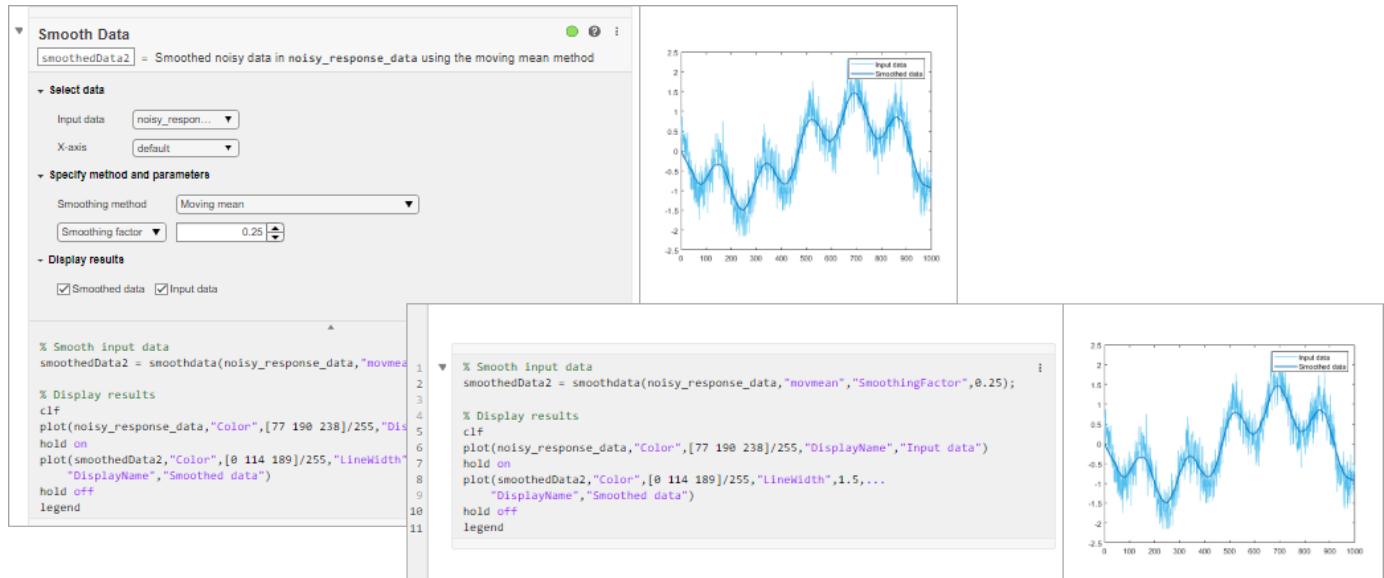


You can use the resulting output argument in subsequent code, including as inputs to additional Live Editor tasks.

View and Edit Generated Code

To see the MATLAB commands that the task runs, click the options button at the top-right of the task and select either **Controls and Code** or **Code Only**. You also can use the down arrow at the bottom of the task to show and hide the generated code. The generated code is read-only.

To edit the generated code, click the options button and select **Convert Task to Editable Code**. This option removes the task and replaces it with the generated code, which you then can edit.



See Also

More About

- “Add Interactive Controls to a Live Script” on page 19-47
- “Clean Messy Data and Locate Extrema Using Live Editor Tasks”
- MATLAB Live Script Gallery

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**.

Open Existing Function as Live Function

If you have an existing function, you can open it as a live function in the Live Editor. 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 script to formatted content in the 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**.

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.

Create Live Function from Selected Code

If you have an existing large live script or function, you can break it into smaller pieces by automatically converting selected areas of code into functions or local functions. This is called code refactoring.

To refactor a selected area of code, select one or more lines of code and on the **Live Editor** tab, in the **Code** section, click  **Refactor**. Then, select from the available options. MATLAB creates a function with the selected code and replaces the original code with a call to the newly created function.

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 19-62.

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 Live Function

You can run live functions using several methods, including calling them from the Command Window or calling them from a live script. In MATLAB Online, you also can use the  Run button.

To run a live function from the Command Window, 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.

```

1 x = 1:10;
2 n = length(x);
3 avg = mymean(x,n);
4 disp(['Average = ', num2str(avg)])

```

Average = 5.5

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 x = 1:10; 2 n = length(x); 3 avg = mymean(x,n); 4 disp(['Average = ', num2str(avg)]) </pre>	<pre> a = 5.5 Average = 5.5 </pre>
---	------------------------------------

In MATLAB Online, you can use the  **Run** button to run live functions interactively. When you run a live function using the  **Run** button, the output displays in the Command Window. To run live functions that require input argument values or any other additional setup, configure the  **Run** button by clicking **Run+** and adding one or more commands. For more information about configuring the  **Run** button, see “Configure the Run Button for Functions” on page 20-7.

Save Live Functions as Plain Code

To save a live 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 (UTF-8) (*.m)** as the **Save as type**.
- 3 Click **Save**.

When saving, MATLAB converts all formatted content to publish markup.

See Also

More About

- “Add Help for Live Functions” on page 19-62

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 a MATLAB live script window. The help text is displayed above the code, and the code itself is numbered from 1 to 10. The code defines a function `addme` that adds two numbers or a number to itself based on the input count.

```
1 Add two values together
2 C = ADDME(A) adds A to itself.
3 C = ADDME(A,B) adds A and B together.
4 See also SUM, PLUS.

function c = addme(a,b)
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 20-37. 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 19-32.

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 switch nargin
4   case 2
5     c = a + b;
6   case 1
7     c = a + a;
8   otherwise
9     c = 0;
10 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.

You also can add help to live functions by inserting comments at the beginning of the file. Comments at the beginning of the file display as help text when you use the `help` and `doc` commands, similar to how text at the beginning of the file displays. For more information about adding help using comments, see “Add Help for Your Program” on page 20-5.

See Also

More About

- “Create Live Functions” on page 19-59
- “Format Files in the Live Editor” on page 19-32
- “Share Live Scripts and Functions” on page 19-66

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, Microsoft Word, 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	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. MATLAB supports live scripts in Versions R2016a and above, and live functions in Versions R2018a and above.
As a full screen presentation	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. 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.
With users of previous MATLAB versions	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. For more information, see “Save Live Scripts as Plain Code” on page 19-11.

Ways to Share	Instructions
As a static document capable of being viewed outside of MATLAB	<p>Export the live script or function to a standard format. Available formats include PDF, Microsoft Word, HTML, and LaTeX.</p> <p>To export your live script or function to one of these formats, on the Live Editor tab, select Export > Export to PDF, Export > Export to Word, Export > Export to HTML, or Export > Export to LaTeX. The saved file closely resembles the appearance of the live script or function when viewed in the Live Editor with output inline.</p> <p>To export all of the live scripts and live functions in a folder, on the Live Editor tab, select Export > Export Folder.</p> <p>To export a live script or function in MATLAB Online, on the Live Editor tab, select Save and then select from the available options.</p> <p>When exporting to LaTeX, MATLAB creates a separate <code>matlab.sty</code> file in the same folder as the output document, if one does not exist already. <code>.sty</code> files, also known as LaTeX Style Documents, give you more control over the appearance of the output document.</p> <p>To customize the format of figures as well as the document paper size, orientation, and margins before exporting, use settings. For more information, see <code>matlab.editor Settings</code>.</p>

Hide Code Before Sharing

Consider hiding the code in the live script before sharing it as an interactive or static document. When the code is hidden, the Live Editor only displays labeled controls, output, and formatted text.

To hide the code, click the  hide code button to the right of the live script. You also can go to the **View** tab, and in the **View** section, click  **Hide Code**. To show the code again, click the  output inline button or the  output on right button.

See Also

Related Examples

- “Create Live Scripts in the Live Editor” on page 19-6
- “Format Files in the Live Editor” on page 19-32

- MATLAB Live Script Gallery

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 33-14 or “Register Binary Files with Git” on page 33-24.

See Also

More About

- “What Is a Live Script or Function?” on page 19-2
- “Create Live Scripts in the Live Editor” on page 19-6

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, click **New 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.

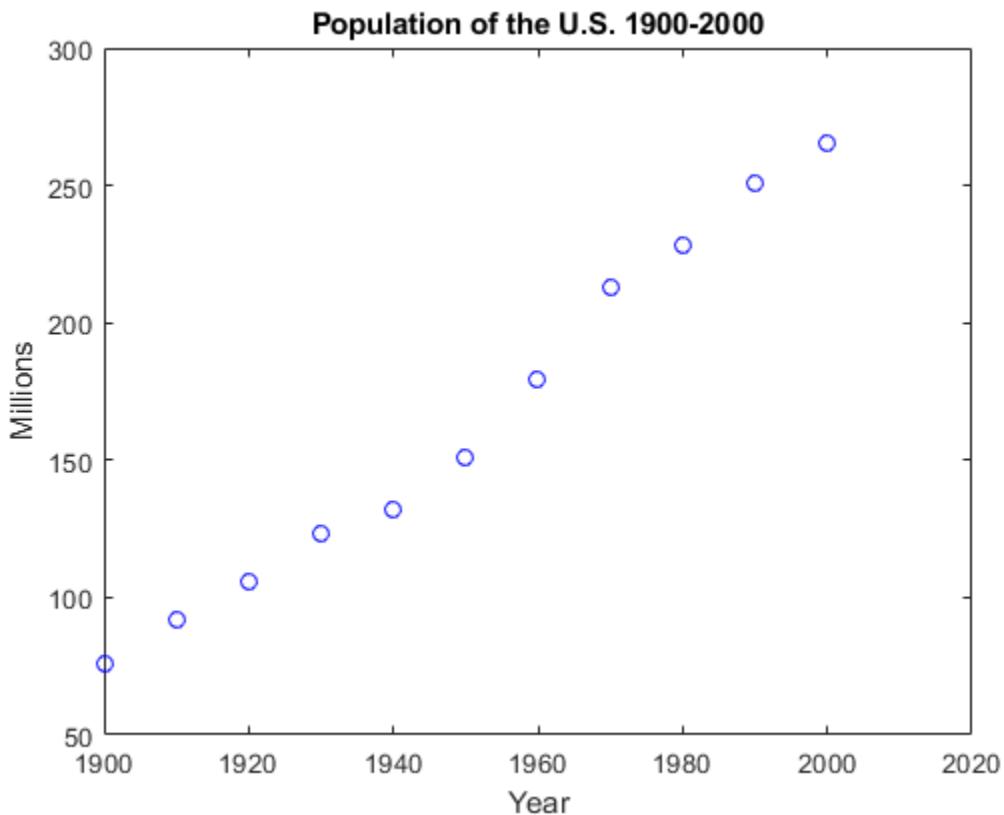
```
years = (1900:10:2000); % Time interval
pop = [75.995 91.972 105.711 123.203 131.669 ...
    150.697 179.323 213.212 228.505 250.633 265.422] % Population Data
pop = 1×11
    75.9950    91.9720   105.7110   123.2030   131.6690   150.6970   179.3230   213.2120   228.5050   250.6330
```

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');
axis([1900 2020 0 400]);
title('Population of the U.S. 1900-2000');
ylabel('Millions');
xlabel('Year');
ylim([50 300]) % Plot the population data
```



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:

$$\begin{array}{ll} y = ax + b & \text{linear} \\ y = ax^2 + bx + c & \text{quadratic} \\ y = ax^3 + bx^2 + cx + d. & \text{cubic} \end{array}$$

```
x = (years-1900)/50;
coef1 = polyfit(x,pop,1)
```

```
coef1 = 1x2
```

```
98.9924    66.1296
```

```
coef2 = polyfit(x,pop,2)
```

```
coef2 = 1x3
```

```
15.1014    68.7896    75.1904
```

```
coef3 = polyfit(x,pop,3)
coef3 = 1x4
-17.1908    66.6739    29.4569    80.1414
```

Plotting the Curves

Create sections with any number of text and code lines.

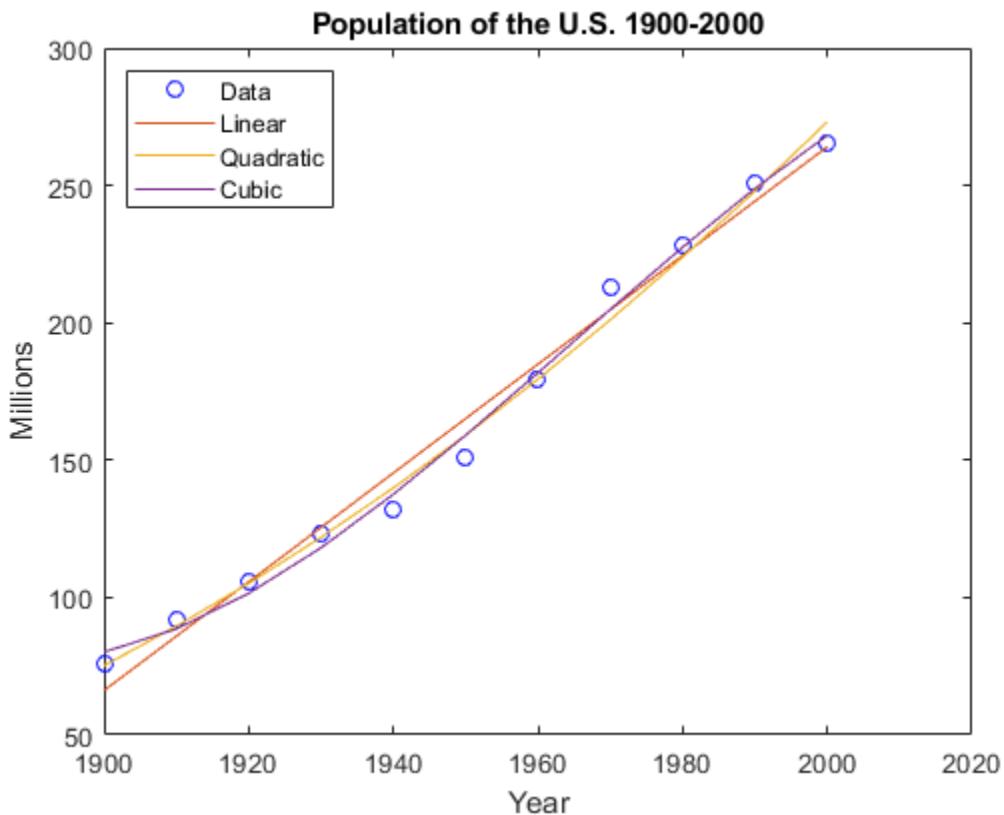
We can plot the linear, quadratic, and cubic curves fitted to the data. We'll use the `polyval` function to evaluate the fitted polynomials at the points in `x`.

```
pred1 = polyval(coef1,x);
pred2 = polyval(coef2,x);
pred3 = polyval(coef3,x);
[pred1; pred2; pred3]

ans = 3x11
66.1296    85.9281    105.7266    125.5250    145.3235    165.1220    184.9205    204.7190    224.5174    244.3159
75.1904    89.5524    105.1225    121.9007    139.8870    159.0814    179.4840    201.0946    223.9134    247.9113
80.1414    88.5622    101.4918    118.1050    137.5766    159.0814    181.7944    204.8904    227.5441    248.9113
```

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 that they can reproduce your results. You also can publish your results as PDF, Microsoft® Word, or HTML documents. Add controls to your live scripts to show users how important parameters affect the analysis. To add controls, go to the **Live Editor** tab, click the **Control** button, and select from the available options.

We can now calculate the predicted population of a given year using our three equations.

```
year = 2018 ;  
xyear = (year-1900)/50;  
pred1 = polyval(coef1,xyear);  
pred2 = polyval(coef2,xyear);  
pred3 = polyval(coef3,xyear);  
[pred1 pred2 pred3]  
  
ans = 1x3  
  
299.7517 321.6427 295.0462
```

For the year 2010 for example, 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.

See Also

More About

- “Create Live Scripts in the Live Editor” on page 19-6
- MATLAB Live Script Gallery

Accelerate Exploratory Programming Using the Live Editor

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 using controls.
- 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    vehicleMile
    _____      _____      _____      _____      _____      _____
    Wyoming        -107.56    43.033     164    380.18    671.53    9261
    District_of_Columbia   -77.027    38.892      43    349.12    240.4    3742
    Vermont        -72.556    44.043      98    550.46    551.52    7855
    North_Dakota    -99.5    47.469     100    461.78    721.84    7594
    South_Dakota    -99.679    44.272     197    563.3    882.77    8784
    Delaware        -75.494    39.107     134    533.94    728.52    9301
    Montana        -110.58    46.867     229    712.88    1056.7    11207
    Rhode_Island    -71.434    41.589      83    741.84    834.5    8473
    New_Hampshire    -71.559    43.908     171    985.77    1244.6    13216
    Maine          -69.081    44.886     194    984.83    1106.8    14948
```

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

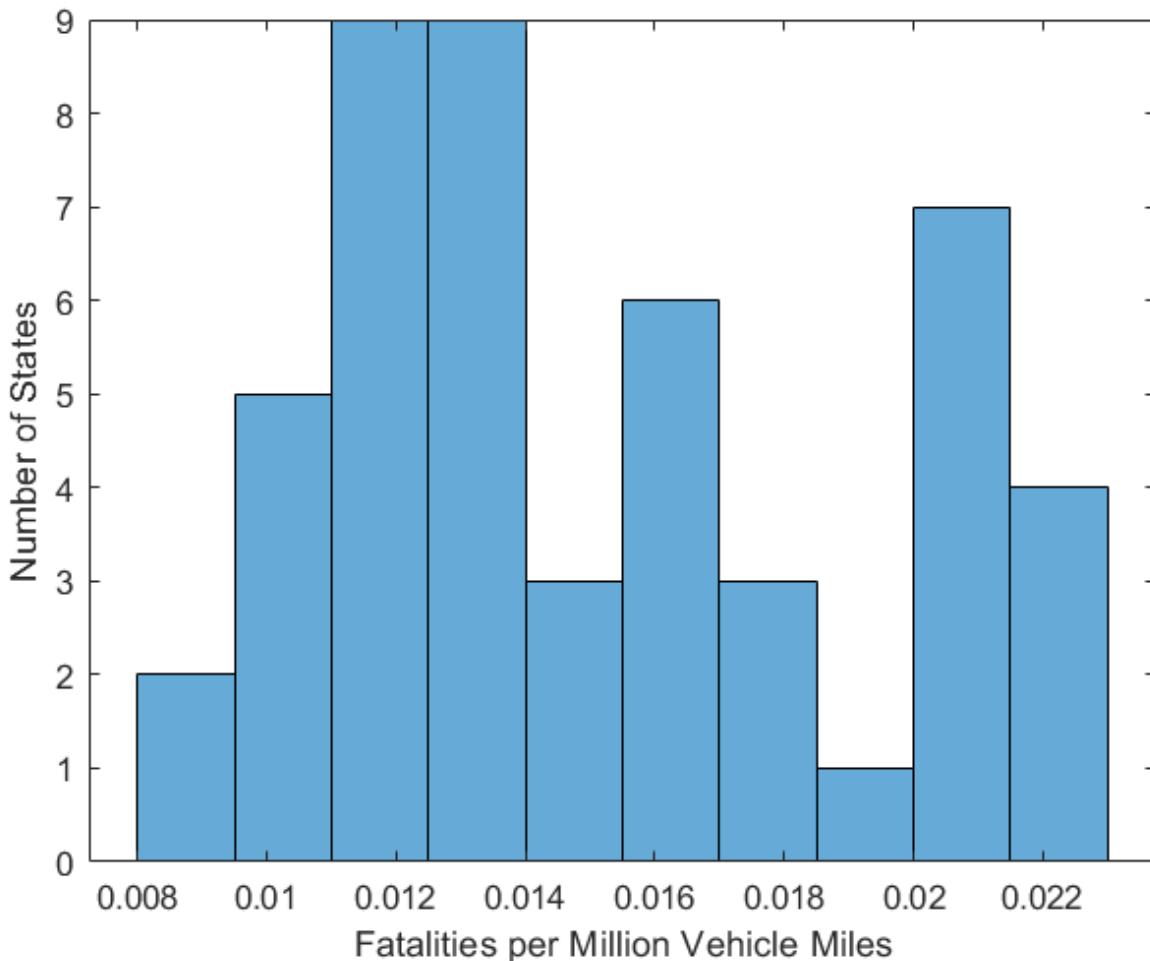
```
disp([states{maxIdx} ' has the highest fatality rate at ' num2str(rate(maxIdx))])  
Mississippi has the highest fatality rate at 0.022825
```

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')
```



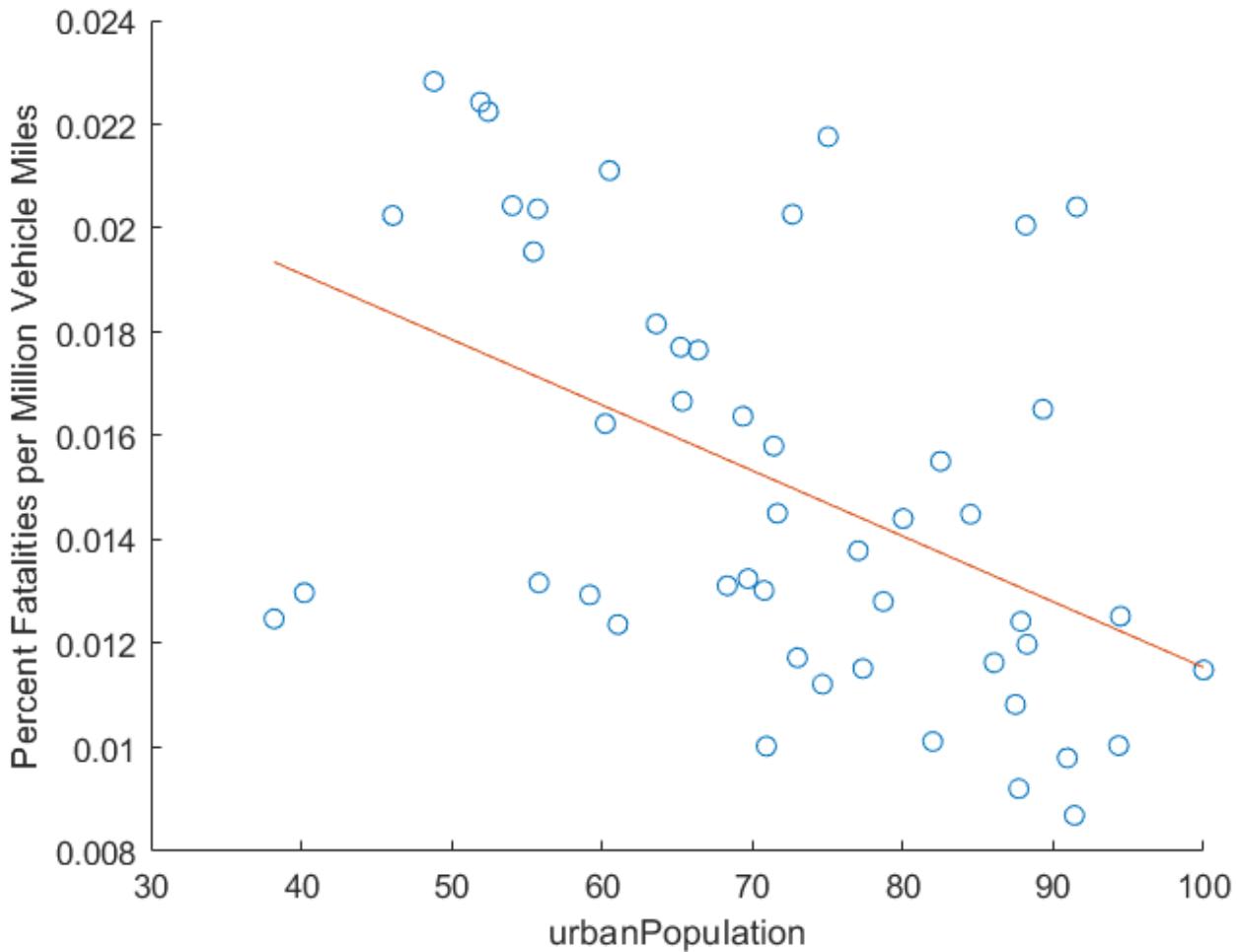
Find Correlations in the Data

You can explore your data quickly in the Live Editor by experimenting with parameter values to see how your results will change. Add controls to change parameter values interactively. To add controls, go to the **Live Editor** tab, click the **Control** button, and select from the available options.

We can experiment with the data to see if any of the variables in the table are correlated with highway fatalities. For example, it appears that highway fatality rates are lower in states with a higher percentage urban population.

```
dataToPlot =  ;
close % Close any open figures
scatter(fatalities.(dataToPlot),rate) % Plot fatalities vs. selected variable
xlabel(dataToPlot)
ylabel('Percent Fatalities per Million Vehicle Miles')

hold on
xmin = min(fatalities.(dataToPlot));
xmax = max(fatalities.(dataToPlot));
p = polyfit(fatalities.(dataToPlot),rate,1); % Calculate & plot least squares line
plot([xmin xmax], polyval(p,[xmin xmax]))
```

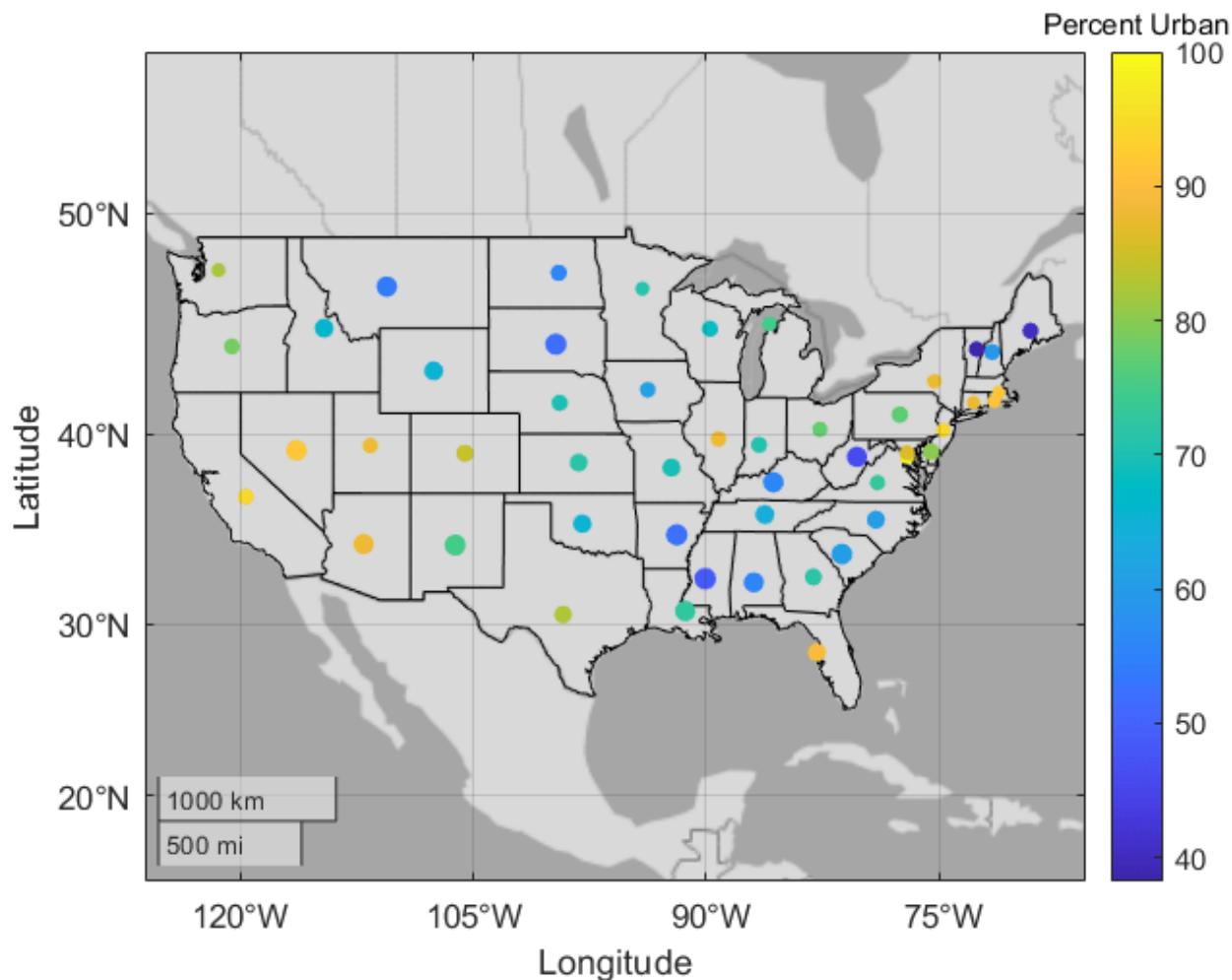


Plot Fatalities and Urbanization on a US Map

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, Microsoft® Word, or PDF documents 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')
geobasemap darkwater
hold on
geoscat(fatalities.latitude,fatalities.longitude,2000*rate,fatalities.urbanPopulation,'filled'
c = colorbar;
title(c,'Percent Urban')
```



See Also

More About

- “Create Live Scripts in the Live Editor” on page 19-6
- MATLAB Live Script Gallery

Create an Interactive Narrative with the Live Editor

The following is an example of how to create an interactive narrative in the Live Editor. An interactive narrative ties together the computations that you use to solve a problem. This example shows how to:

- Describe your approach using formatted text
- Show output together with your MATLAB code.
- Describe the underlying mathematics with equations.
- Illustrate important points with images.
- Links to background material.
- Modify parameters and re-run the analysis with controls.
- Plot data for visualization.
- Invite colleagues to extend your analysis.

Overall Approach

Include formatted text as part of the interactive narrative. Use bold, italic, and underlined text to highlight important words. Use bullets or numbers to format lists.

Estimate the *power output* from a typical solar panel installation on a specific date, time, and location by calculating 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

Use the results of these calculations to plot solar and panel radiation for the example day and location. Then, plot the expected panel power generation over the course of a year. To streamline the analysis, use two MATLAB functions created for this example: `solarCorrection` and `panelRadiation`.

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. This in turn depends on the sun's position relative to the panel as the sun moves across the sky. For example, suppose that you want to calculate power output for a solar panel on June 1st at 12 noon in Boston, Massachusetts.

```
lambda = -71.06; % longitude
phi = 42.36; % latitude
UTCoff = -5; % UTC offset
january1 = datetime(2019,1,1);
localTime = datetime(2019,6,1,12,0,0) % January 1st
% Noon on June 1
```

```
localTime = datetime
01-Jun-2019 12:00:00
```

To calculate the sun's position for a given date and time, use *solar time*. Twelve noon solar time is the time when the sun is highest in the sky. To calculate solar time, 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 earth's orbital eccentricity and axial tilt.

Calculate solar time using the `solarCorrection` function.

```
d = caldays(between(january1,localTime,'Day')); % Day of year
solarCorr = solarCorrection(d,lambda,str2double(UTCoff)); % Correction to local time
solarTime = localTime + minutes(solarCorr)

solarTime = datetime
01-Jun-2019 12:18:15
```

Solar Declination and Elevation

Include equations to describe the underlying mathematics. Create equations using LaTeX commands. To add a new equation, go to the **Insert** tab and click the **Equation** button. Double-click an equation to edit it in the Equation Editor.

The solar 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. Calculate the solar declination for a given day of the year (d) using the equation

$$\delta = \sin^{-1} \left(\sin(23.45) \sin \left(\frac{360}{365} (d - 81) \right) \right)$$

Then, use the declination (δ), the latitude (ϕ), and the *hour angle* (ω) to calculate the sun's elevation (α) at the current time. The hour angle is the number of degrees of rotation of the earth between the current solar time and solar noon.

```
 $\alpha = \sin^{-1}(\sin\delta\sin\phi + \cos\delta\cos\phi\cos\omega)$ 

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) + ...
             cosd(delta)*cosd(phi)*cosd(omega));
disp(['Solar Declination = ' num2str(delta) ' Solar Elevation = ' num2str(alpha)])
```

Solar Declination = 21.8155 Solar Elevation = 69.113

Calculate the time of sunrise and sunset in Standard Time using the sun's declination and the local latitude.

$$\text{sunset} = 12 + \frac{\cos^{-1}(-\tan\phi\tan\delta)}{15^\circ} - \frac{TC}{60}$$

$$\text{sunrise} = 12 - \frac{\cos^{-1}(-\tan\phi\tan\delta)}{15^\circ} - \frac{TC}{60}$$

```
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')])

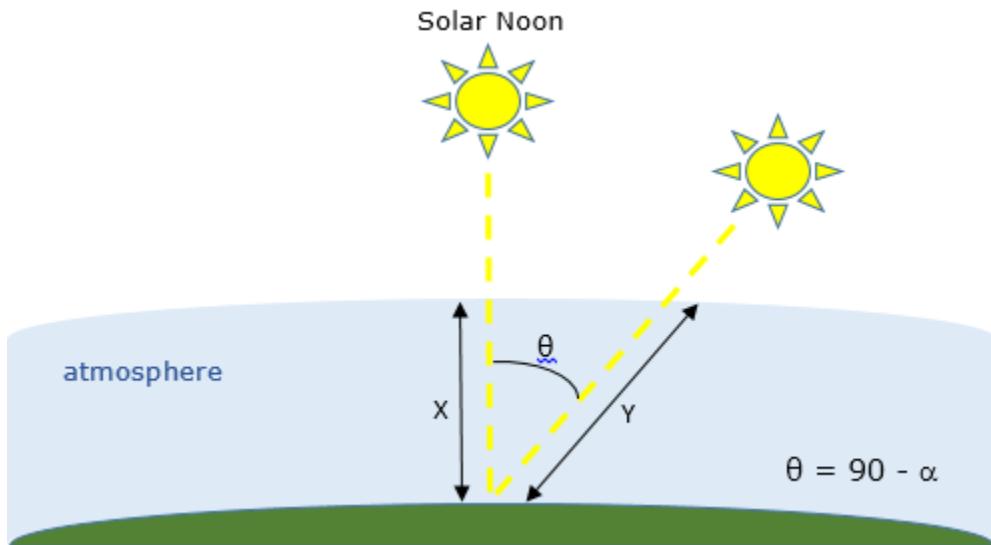
Sunrise = 04:16:06    Sunset = 19:07:22

```

Air Mass and Solar Radiation

Include images to illustrate important points in your story. To include an image, copy and paste an image from another source or go to the **Insert** tab and click the **Image** button.

As light from the sun passes through the earth's atmosphere, some of the solar radiation is absorbed. Air mass is 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°, as shown in the diagram below. It is a function of solar elevation (α).



The larger the air mass, the less radiation reaches the ground. Calculate the air mass using the equation

$$AM = \frac{1}{\cos(90 - \alpha) + 0.5057(6.0799 + \alpha)^{-1.6364}}.$$

Then, calculate the solar radiation reaching the ground (in kilowatts per square meter) using the empirical equation

$$sRad = 1.353 * 0.7^{AM^{0.678}}.$$

```

AM = 1/(cosd(90-alpha) + 0.50572*(6.07955+alpha)^-1.6354);
solarRad = 1.353*0.7^(AM^0.678); % kW/m^2
disp(['Air Mass = ' num2str(AM) '    Solar Radiation = ' num2str(solarRad) ' kW/m^2'])

```

Air Mass = 1.0698 Solar Radiation = 0.93141 kW/m²

Solar Radiation on Fixed Panels

Use hyperlinks to reference supporting information from other sources. To add a hyperlink, go to the **Insert** 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 also depends on the solar 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 solar azimuth is 180° corresponding to the direction south. Calculate the solar azimuth using the 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 = 191.7888
```

In the northern hemisphere, a typical solar panel installation has panels oriented toward the south with a panel azimuth (β) of 180° . At northern latitudes, a typical tilt angle (τ) is 35° . Calculate the panel radiation for fixed panels from the total solar radiation using the equation

$$pRad = sRad[\cos(\alpha)\sin(\tau)\cos(\beta - \gamma) + \sin(\alpha)\cos(\tau)].$$

```
beta = 180; % Panel azimuth
tau = 35; % Panel tilt
panelRad = solarRad*max(0,(cosd(alpha)*sind(tau)*cosd(beta-gamma) + sind(alpha)*cosd(tau)));
disp(['Panel Radiation = ' num2str(panelRad) ' kW/m^2'])
Panel Radiation = 0.89928 kW/m^2
```

Panel Radiation and Power Generation For a Single Day

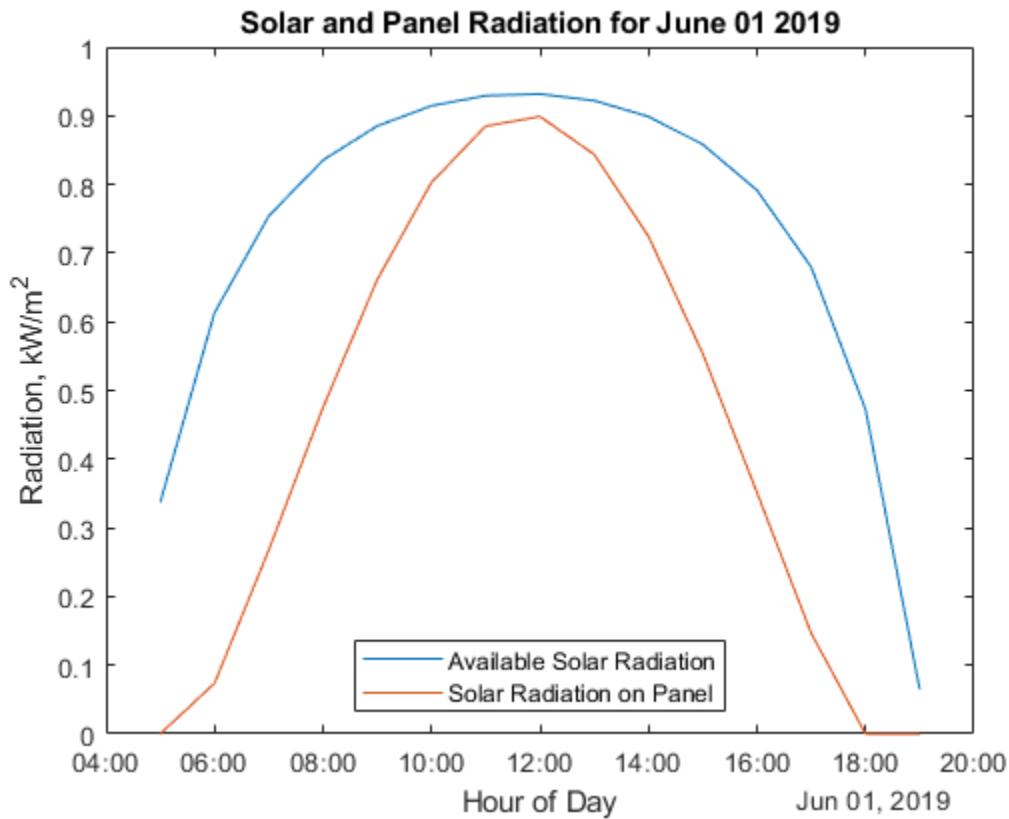
Modify parameters using interactive controls. Display plots together with the code that produced them.

Panel Radiation

For a given day of the year, calculate the total solar radiation and the radiation on the panel. To simplify the analysis, use the `panelRadiation` function. Try different dates to see how the solar and panel radiation change depending on the time of year.

```
selectedMonth = June;
selectedDay = 1;
selectedDate = datetime(2019,selectedMonth,selectedDay);
[times,solarRad,panelRad] = panelRadiation(selectedDate,lambda,phi,UTCoff,tau,beta);
```

```
plot(times,solarRad,times,panelRad)
title(['Solar and Panel Radiation for ' datestr(selectedDate,'mmmm dd yyyy')])
xlabel('Hour of Day');
ylabel('Radiation, kW/m^2')
legend('Available Solar Radiation', 'Solar Radiation on Panel', 'Location', 'South')
```



Power Generation

So far, the calculations assume that all of the radiation reaching the panel is available to generate power. However, solar panels do not convert 100% of available solar radiation into electricity. The efficiency of a solar panel is the fraction of the available radiation that is converted. The efficiency of a solar panel depends on the design and materials of the cell.

Typically, a residential installation includes 20m² of solar panels with an efficiency of 25%. 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
radiation = sum(panelRad(1:end-1)+panelRad(2:end))/2;
dayPower = eff*pSize*radiation; % Panel electric output in kW
disp(['Expected daily electrical output for ' datestr(selectedDate) '=' num2str(dayPower) ' kW'])
```

Expected daily electrical output for 01-Jun-2019 = 33.4223 kW-hrs

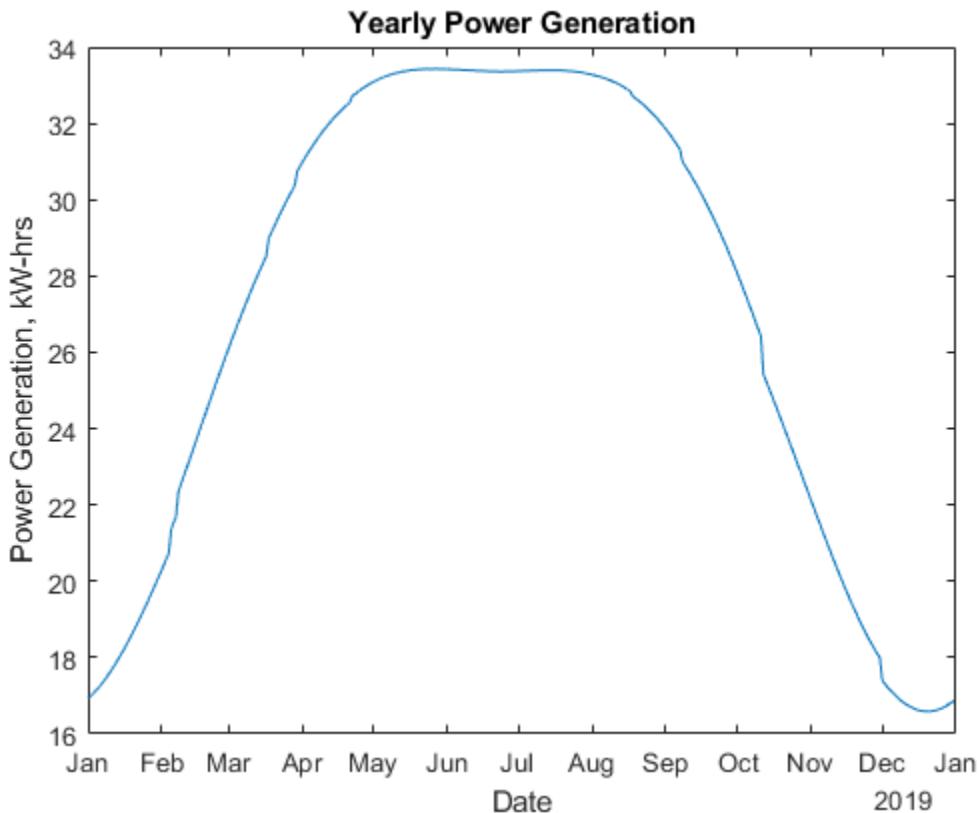
Power Generation for the Whole Year

Hover over a plot to interact with it. Interacting with a plot in the Live Editor will generate code that you can then add to your script.

Repeat the calculation to estimate power generation for each day of the year.

```
yearDates = datetime(2019,1,1:365); % Create a vector of days in the year
dailyPower = zeros(1,365);
for i = 1:365
    [times,solarRad,panelRad] = panelRadiation(yearDates(i),lambda,phi,UTCoff,tau,beta) ;
    radiation = sum(panelRad(1:end-1)+panelRad(2:end))/2;
    dailyPower(i) = eff*pSize*radiation;
end

plot(yearDates,dailyPower)
title('Yearly Power Generation')
xlabel('Date');
ylabel('Power Generation, kW-hrs')
```



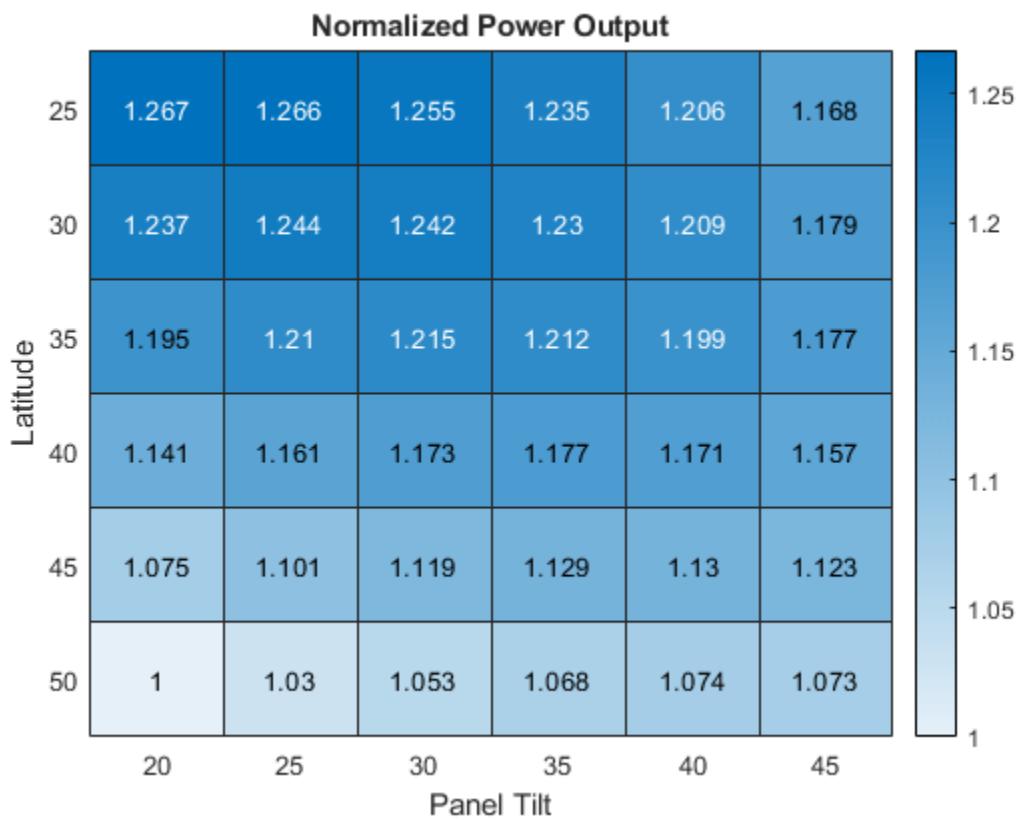
```
yearlyPower = sum(dailyPower);
disp(['Expected annual power output = ' num2str(yearlyPower) ' kW-hrs'])
```

Expected annual power output = 9954.3272 kW-hrs

Panel Tilt and Latitude

Use a heatmap to determine how panel tilt affects power generation. The heatmap below shows that the optimal panel tilt for any location is about 5° less than the latitude.

```
load LatitudeVsTilt.mat
heatmap(powerTbl,'Tilt','Latitude',...
    'ColorVariable','Power');
xlabel('Panel Tilt')
ylabel('Latitude')
title('Normalized Power Output')
```



Extend 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 is 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, you can use data from these government websites.

- Use historical local weather data from the National Weather Service website.

- Use measured solar radiation data from the National Solar Radiation Database.

See Also

More About

- “Create Live Scripts in the Live Editor” on page 19-6
- “Format Files in the Live Editor” on page 19-32
- MATLAB Live Script Gallery

Create Interactive Course Materials Using the Live Editor

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 **Insert** 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')
```

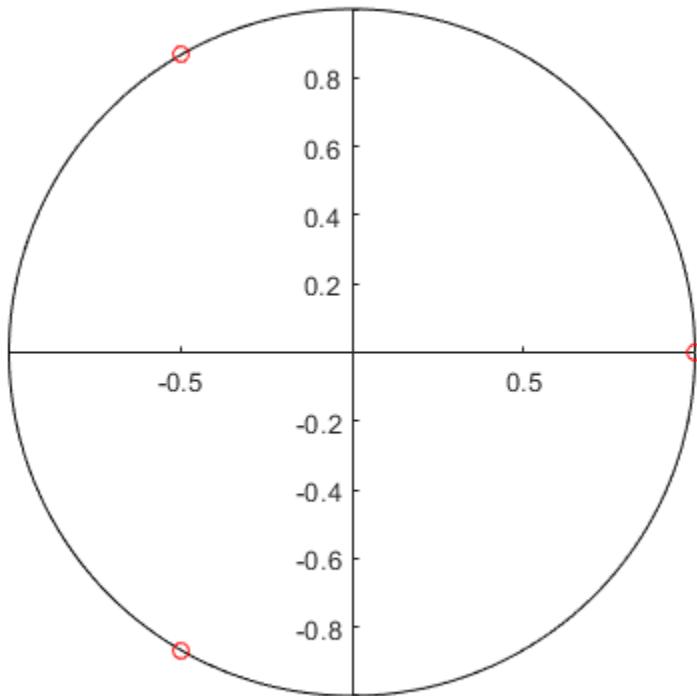
```
1.0000 + 0.0000i
-0.5000 - 0.8660i
-0.5000 + 0.8660i
```

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 **Insert** 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. Add controls to show students how important parameters affect the analysis. To add controls, go to the **Live Editor** tab, click the **Control** button, and select from the available options.

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 :

```
n = 6; % Input control
roots = zeros(1, n);
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
0.5000 - 0.8660i
-0.5000 - 0.8660i
-1.0000 - 0.0000i
-0.5000 + 0.8660i
0.5000 + 0.8660i

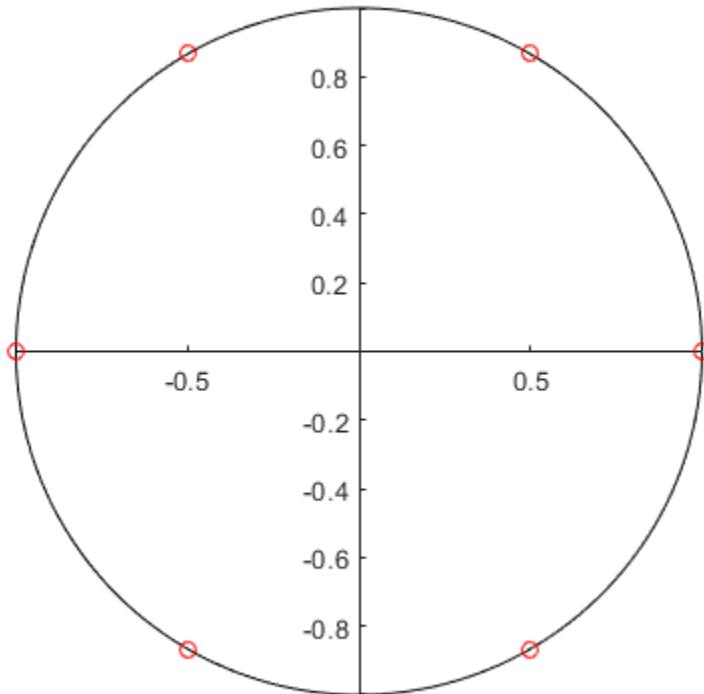
```

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

```

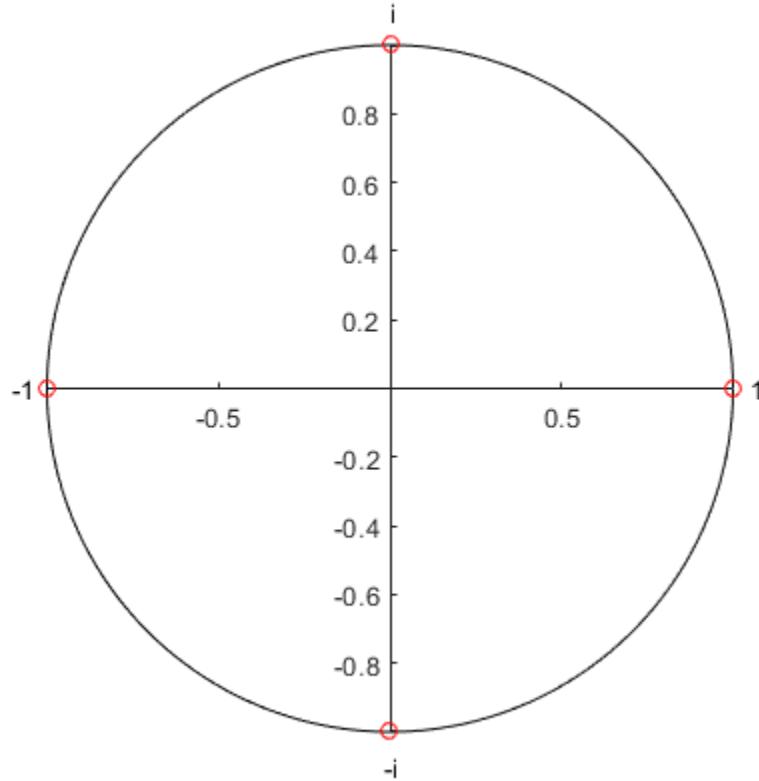


Finding the n th roots of -1 , i , and $-i$

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')
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)

See Also

More About

- “Create Live Scripts in the Live Editor” on page 19-6
- MATLAB Live Script Gallery

Create Examples Using the Live Editor

This example shows how to use the Live Editor to document or create an example for your code. The example uses formatted text, equations, and runnable code to document the sample function `estimatePanelOutput`. It also uses sliders and text fields to allow users to experiment with different function inputs.

To view and interact with the controls, open this example in MATLAB®.

`estimatePanelOutput` Function

Overview

The `estimatePanelOutput` function estimates the power output from a typical solar panel installation based on location, panel size, and panel efficiency.

The function uses this formula to calculate the solar declination:

$$\alpha = \sin^{-1} \left(\sin(23.45) \sin\left(\frac{360}{365}(d - 81)\right) \right)$$

and this formula to calculate solar elevation:

$$\epsilon = \sin^{-1} (\sin \delta \sin \phi + \cos \delta \cos \phi \cos \omega)$$

The function calls two additional MATLAB functions, `solarCorrection` and `hourlyPanelRadiation`.

Examples

Estimate Panel Output with Standard Efficiency

Call the `estimatePanelOutput` function with the panel size, specified in m^2 . Since no efficiency value is specified, `calculatePanelOutput` uses the default efficiency value of 25%.

Specify the latitude, longitude, and UTC offset for the location of your panels, as well as the size of your panels in m^2 .

```
longitude = -71.06;
latitude = 42.36;
UTCoffset = -5;
pSize = 10;
```

Calculate the expected electrical output of the panel.

```
panelOutput = estimatePanelOutput(longitude, latitude, UTCoffset, pSize);
disp(['Expected electrical output = ' num2str(panelOutput) ' kW'])

Expected electrical output = 2.2472 kW
```

See Also

More About

- “Create Live Scripts in the Live Editor” on page 19-6
- “Add Help for Live Functions” on page 19-62
- “Display Custom Documentation” on page 31-21
- MATLAB Live Script Gallery

Create an Interactive Form Using the Live Editor

This example shows how to use the Live Editor to create an interactive form that completes a calculation based on input provided by a user. The example uses drop-down menus and sliders to prompt the user for input, and then uses a button to run a calculation using the provided input. The example hides the code to only show the controls and results to the user.

To view and interact with the example, open it in MATLAB®. To view the code for this example in MATLAB, go to the **View** tab and click **Output Inline** or **Output on Right**.

Solar Panel Output Estimator

Specify the location of your panels.

Location:

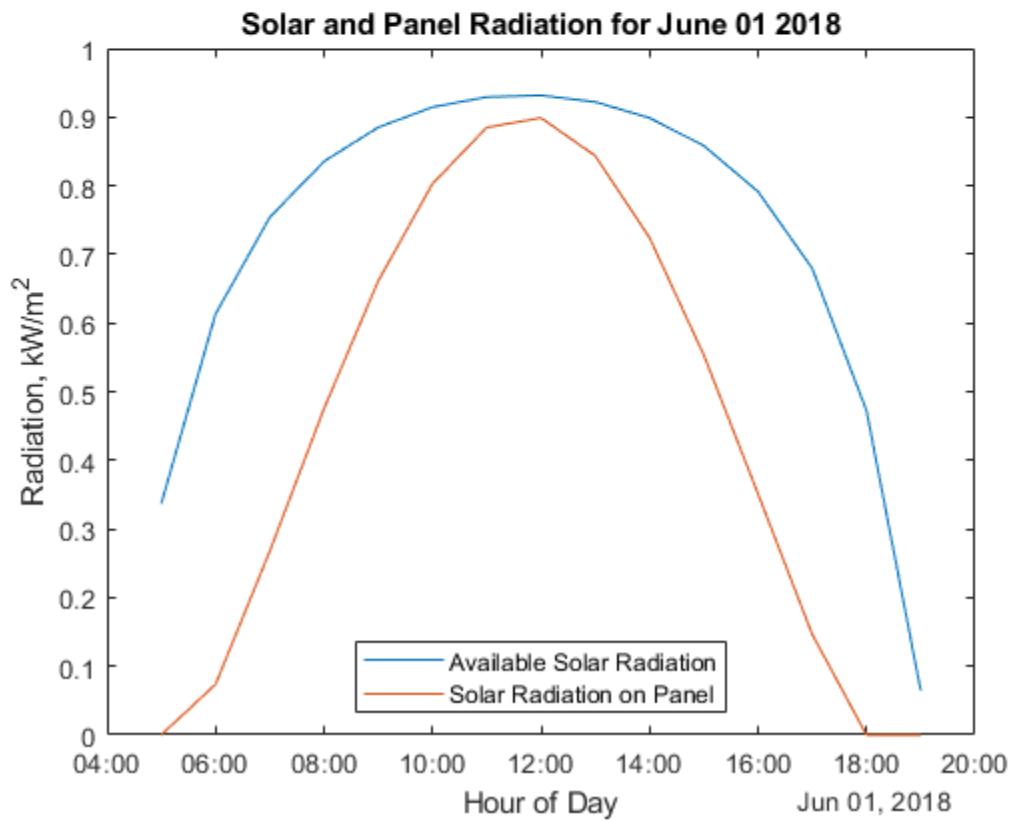
Specify the panel size and efficiency value.

Panel Size (m^2):

Panel Efficiency:

Results:

Expected electrical output = 337.0771 kW



See Also

More About

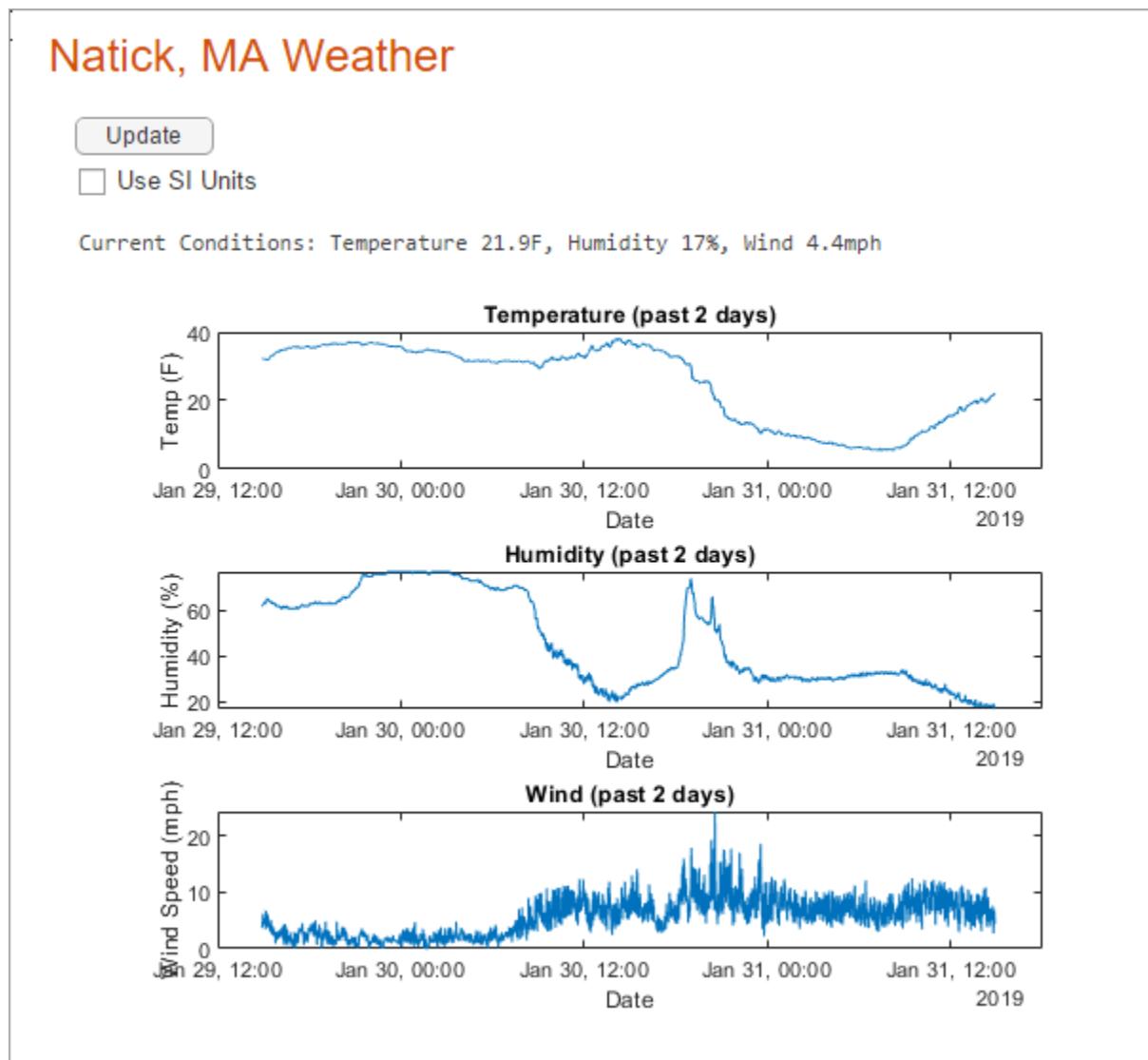
- “Create Live Scripts in the Live Editor” on page 19-6

- MATLAB Live Script Gallery

Create a Real-time Dashboard Using the Live Editor

This example shows how to use the Live Editor to create a dashboard for displaying and analyzing real-time data. The example uses a button and a check-box to get and display real-time data on demand. The example hides the code to only show the controls and results to the user.

To view and interact with the example, open it in MATLAB®. To view the code for this example in MATLAB, go to the **View** tab and click **Output Inline** or **Output on Right**.



See Also

More About

- “Create Live Scripts in the Live Editor” on page 19-6
- MATLAB Live Script Gallery

Acknowledgments

MATLAB Live Editor uses Antenna House® XSL Formatter. Antenna House is a trademark of Antenna House, Inc.

Antenna House XSL Formatter© 2009-2019 Copyright Antenna House, Inc.

Function Basics

- “Create Functions in Files” on page 20-2
- “Add Help for Your Program” on page 20-5
- “Configure the Run Button for Functions” on page 20-7
- “Base and Function Workspaces” on page 20-9
- “Share Data Between Workspaces” on page 20-10
- “Check Variable Scope in Editor” on page 20-14
- “Types of Functions” on page 20-17
- “Anonymous Functions” on page 20-20
- “Local Functions” on page 20-25
- “Nested Functions” on page 20-27
- “Resolve Error: Attempt to Add Variable to a Static Workspace.” on page 20-33
- “Private Functions” on page 20-36
- “Function Precedence Order” on page 20-37
- “Update Code for R2019b Changes to Function Precedence Order” on page 20-40
- “Indexing into Function Call Results” on page 20-46

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 instance, 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.

<code>function</code> 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><code>function myOutput = myFunction(x)</code></pre> <p>If your function returns more than one output, enclose the output names in square brackets.</p> <pre><code>function [one,two,three] = myFunction(x)</code></pre> <p>If there is no output, you can omit it.</p> <pre><code>function myFunction(x)</code></pre> <p>Or you can use empty square brackets.</p> <pre><code>function [] = myFunction(x)</code></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> <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><code>function y = myFunction(one,two,three)</code></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 20-9
- “Types of Functions” on page 20-17
- “Add Functions to Scripts” on page 18-14

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). If the function contains an `arguments` block, you also can position the help text immediately below the `arguments` block.

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 `` 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 20-37. 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 18-3
- “Create Help for Classes” on page 31-2
- “Create Help Summary Files — `Contents.m`” on page 31-10
- “Check Which Programs Have Help” on page 31-8
- “Display Custom Documentation” on page 31-21
- “Use Help Files with MEX Functions”

Configure the Run Button for Functions

Functions are program files that accept inputs and return outputs. To run functions that require input argument values or any other additional setup from the Editor, configure the  **Run** button.

To configure the **Run** button in the Editor, click **Run**  and add one or more run commands.

For example:

- 1 Create the function `myfunction.m` that performs a calculation using the inputs `x` and `y` and stores the results in `z`.

```
function z = myfunction(x,y)
z = x.^2 + y;
```

- 2 Go to the **Editor** tab and click **Run** . MATLAB displays the list of commands available for running the function.



- 3 Click the last item in the list and replace the text `type code to run` with a call to the function including the required input arguments. For example, enter the text `result = myfunction(1:10,5)` to run `myfunction` with the input arguments `1:10` and `5`, and store the results in the variable `result`. MATLAB replaces the default command with the newly added command.



To run multiple commands at once, enter the commands on the same line. For example, enter the text `a = 1:10; b = 5; result = myfunction(a,b)` to create the variables `a` and `b` and then call `myfunction` with `a` and `b` as the input arguments.

Note If you define a run command that creates a variable with the same name as a variable in the base workspace, the run command variable overwrites the base workspace variable when you run that run command.

- 4 Click the  **Run** button. MATLAB runs the function using the first run command in the list. For example, click  **Run** to run `myfunction` using the command `result = myfunction(1:10,5)`. MATLAB displays the result in the Command Window.

```
result =  
6 9 14 21 30 41 54 69 86
```

To run the function using a different run command from the list, click **Run ▾** and select the desired command. 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, click **Run ▾**, right-click the command, and then select **Edit** or **Delete**.

Note Running live functions in the Live Editor using the  **Run** button is only supported in MATLAB Online. When you run a live function using the  **Run** button, the output displays in the Command Window. To run a live function in an installed version of MATLAB, call the function from the Command Window or from a script or live script.

See Also

More About

- “Calling Functions”
- “Create Functions in Files” on page 20-2
- “Create Live Functions” on page 19-59

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 20-10

More About

- “Nested Functions” on page 20-27

Share Data Between Workspaces

In this section...

- “Introduction” on page 20-10
- “Best Practice: Passing Arguments” on page 20-10
- “Nested Functions” on page 20-10
- “Persistent Variables” on page 20-11
- “Global Variables” on page 20-12
- “Evaluating in Another Workspace” on page 20-12

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 20-27.

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 2-86.

See Also

More About

- “Base and Function Workspaces” on page 20-9

Check Variable Scope in Editor

In this section...

"Use Automatic Function and Variable Highlighting" on page 20-14

"Example of Using Automatic Function and Variable Highlighting" on page 20-14

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 20-28 and "Share Data Between Workspaces" on page 20-10.

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 20-12, persistent variables on page 20-11, and variables within nested functions. (For more information, see "Nested Functions" on page 20-10.)

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:

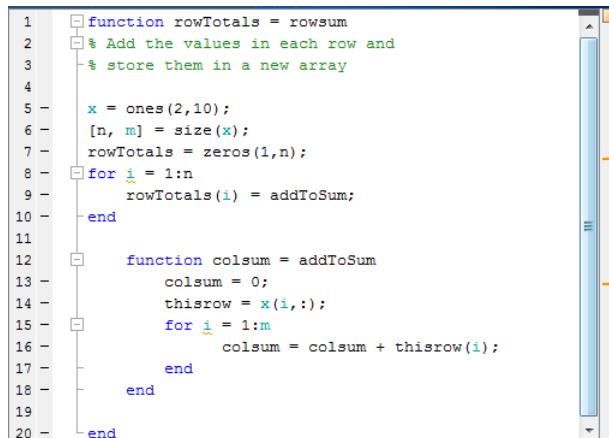
```
ans =
0      0      0      0      0      0      0      0      0      10
```

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
```

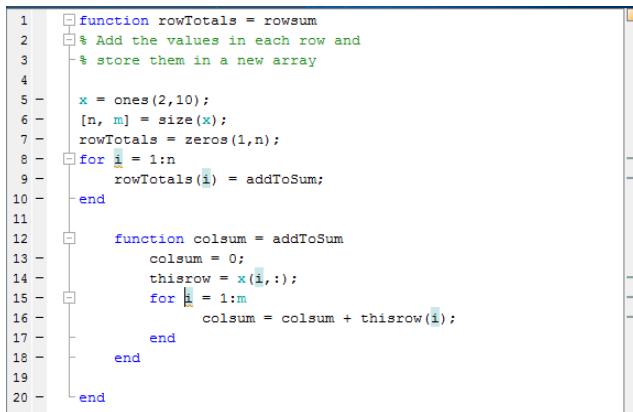
- 3 Hover the mouse pointer over an instance of variable *i*.

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.



The screenshot shows a MATLAB code editor window. 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 k = 1:m
16     colsum = colsum + thisrow(k);
17 end
18 end
19 end
20 end
```

The variable *i* is highlighted in blue throughout the code. In the bottom right corner of the editor, there is an indicator bar with several vertical markers corresponding to the highlighted instances of *i*.

- 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 20-17
- “Private Functions in a Subfolder” on page 20-18
- “Anonymous Functions Without a File” on page 20-18

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.

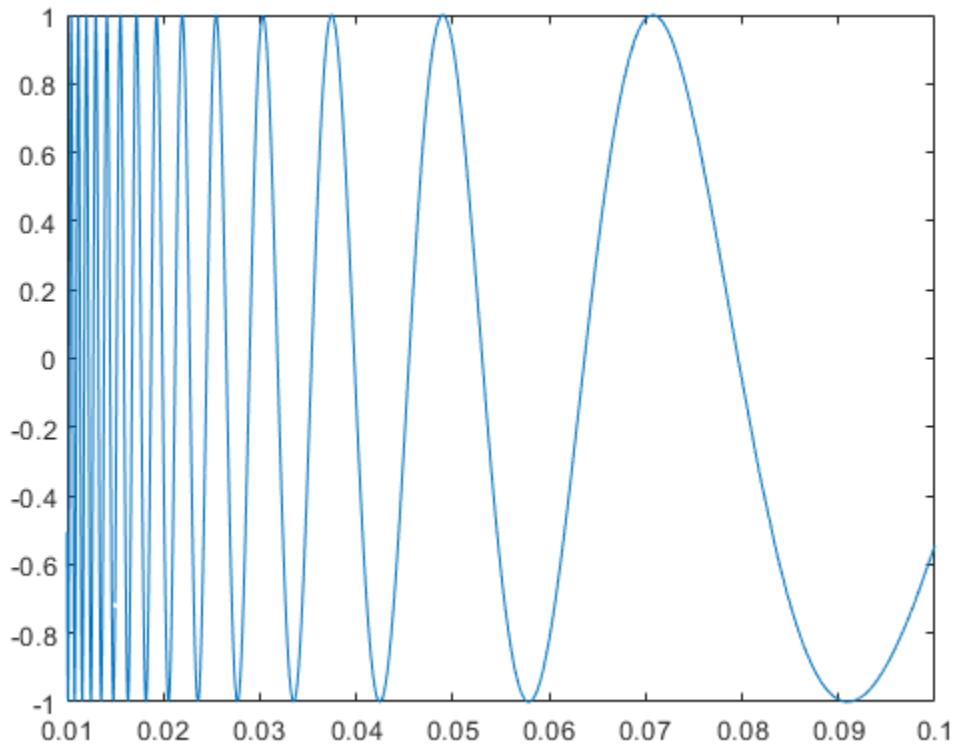
You can use the function handle to evaluate the function for particular values, such as

```
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 20-25
- “Nested Functions” on page 20-27
- “Private Functions” on page 20-36
- “Anonymous Functions” on page 20-20

Anonymous Functions

In this section...

- "What Are Anonymous Functions?" on page 20-20
- "Variables in the Expression" on page 20-21
- "Multiple Anonymous Functions" on page 20-21
- "Functions with No Inputs" on page 20-22
- "Functions with Multiple Inputs or Outputs" on page 20-22
- "Arrays of Anonymous Functions" on page 20-23

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 multiple inputs and return one output. 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 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
```

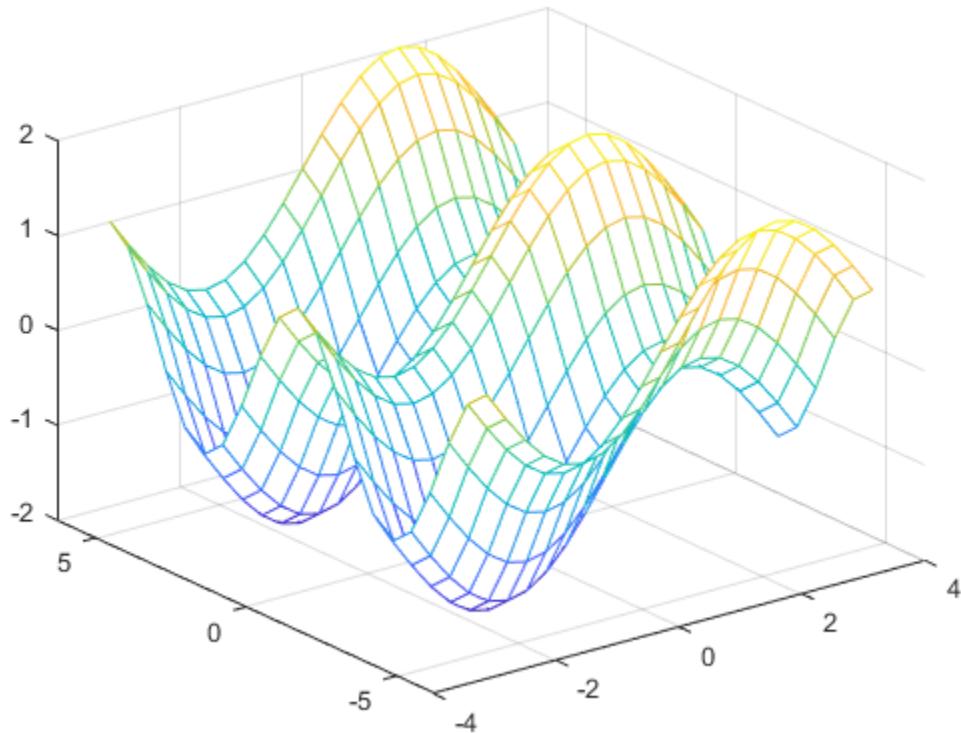
However, an anonymous function returns only one output. If the expression in the function returns multiple outputs, then you can request them when you invoke the function handle.

For example, the `ndgrid` function can return as many outputs as the number of input vectors. This anonymous function that calls `ndgrid` returns only one output (`mygrid`). Invoke `mygrid` to access the outputs returned by the `ndgrid` function.

```
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
- “Types of Functions” on page 20-17

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 18-14.

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 20-27
- “Function Precedence Order” on page 20-37
- “Types of Functions” on page 20-17

Nested Functions

In this section...

- “What Are Nested Functions?” on page 20-27
- “Requirements for Nested Functions” on page 20-27
- “Sharing Variables Between Parent and Nested Functions” on page 20-28
- “Using Handles to Store Function Parameters” on page 20-29
- “Visibility of Nested Functions” on page 20-31

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 “Resolve Error: Attempt to Add Variable to a Static Workspace.” on page 20-33.)

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
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
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);

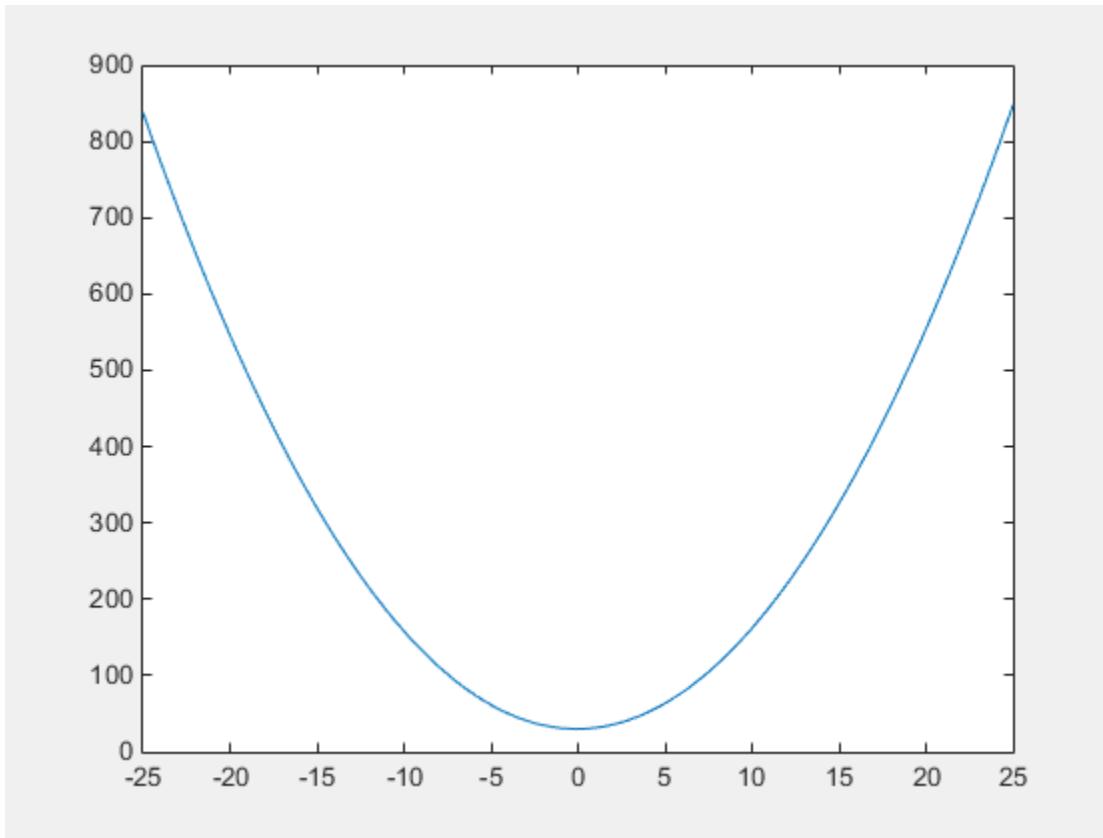
X = 25;
Y = p(X)

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`:

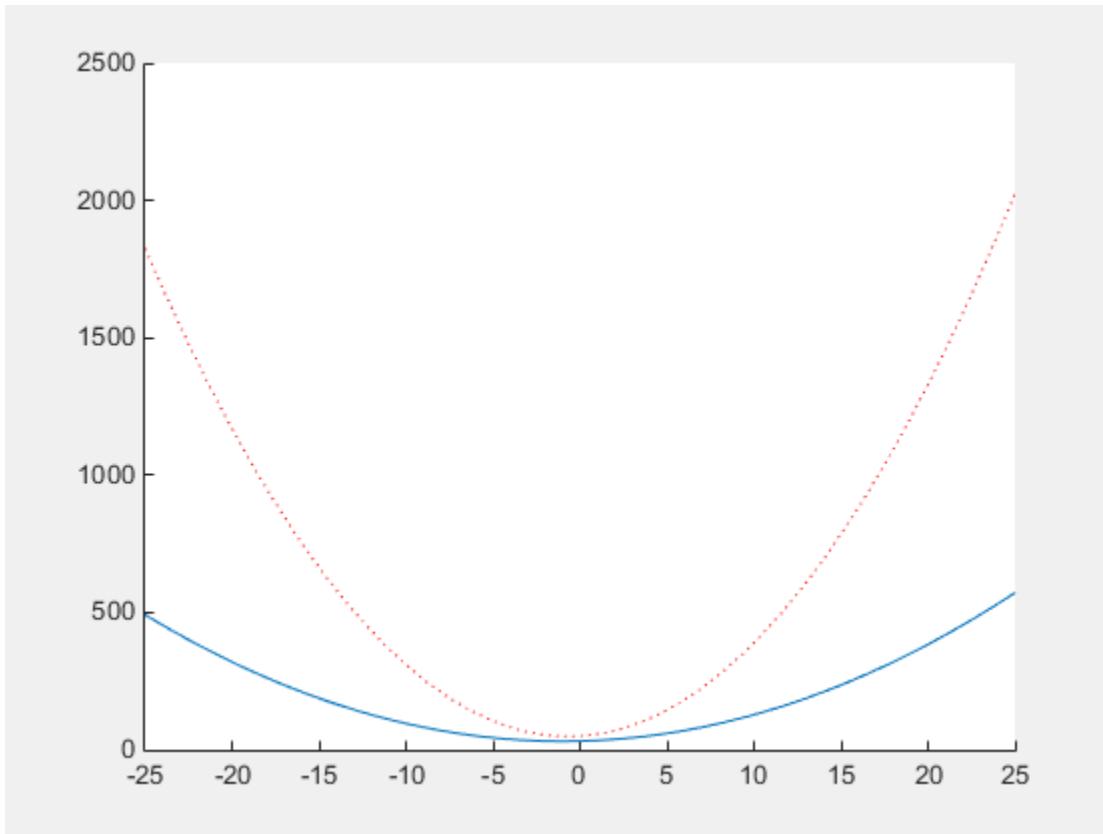
```
fplot(p, [-25,25])
```



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)
    end

```

```
function E(x) % Nested in D
    disp(x)
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 20-29. Only functions that can call a nested function can create a handle to it.

See Also

More About

- “Resolve Error: Attempt to Add Variable to a Static Workspace.” on page 20-33
- “Create Function Handle” on page 13-2
- “Checking Number of Arguments in Nested Functions” on page 21-8
- “Types of Functions” on page 20-17

Resolve Error: Attempt to Add Variable to a Static Workspace.

Issue

The workspaces for nested and anonymous functions are static. This means that all variables used within the function must be present in the text of the code.

If you attempt to dynamically add a variable to the static 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.`

For more information about the differences between the base and function workspaces, see “Base and Function Workspaces” on page 20-9. For more information about nested functions, see “Nested Functions” on page 20-27.

Possible Solutions

Declare Variable in Advance

One way to avoid dynamically adding a variable to static workspaces is to explicitly declare the variable in the code before dynamically assigning a value to that variable. Doing so will cause the variable name to be visible to MATLAB, so the name will be included in the fixed set of variables that make up the static workspace.

For example, suppose a script named `makeX.m` dynamically assigns a value to variable `X`. A function that calls `makeX` and explicitly declares `X` avoids the dynamic adding 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
nestedfx
    function nestedfx
        X = [];
        makeX
    end
end
```

Using eval, evalin, or assignin to Assign New Variables In a Nested Function

Using `eval`, `evalin`, or `assignin` to assign new variables inside of a nested functions will generate an error.

```
function staticWorkspaceErrors
    function nest
        % This will error since x is not declared outside of the eval
        eval("x=2");
    end
end
```

If possible, avoid these functions altogether. See “Alternatives to the eval Function” on page 2-86. If it is not possible to avoid them, then explicitly declare the variable within the parent function:

```
function noStaticWorkspaceErrors
    x = [];
    function nest
        % This will not error since 'x' is declared outside of the eval
        eval("x=2");
    end
end
```

Using a MATLAB script to Assign New Variables In a Nested Function

Calling a MATLAB script that creates a variable inside of a nested function will generate an error. In the example below, the script, `scriptThatIntroducesZ`, contains code that assigns a value to the variable `z`. Since the code does not explicitly declare that `z` is being assigned an error will be thrown.

```
function staticWorkspaceErrors
    function nest
        % This will error since 'z' is not declared outside of this script
        scriptThatIntroducesZ
    end
end
```

To avoid an error, declare the variable within the function before calling the script that assigns a value to it.

```
function noStaticWorkspaceErrors
    function nest
        % This will not error since 'z' is declared outside of the script
        z = [];
        scriptThatIntroducesZ
    end
end
```

Alternatively, convert the script to a function and make `z` its output argument. This approach also makes the code clearer.

Use Explicit Variable Name with load Function

Using `load` to assign variables inside of a nested function, without explicitly specifying the variable name will generate an error. In the example below, `load` is used to load a MAT-file containing the variable `Y`. Since the code does not explicitly declare that `Y` is being assigned an error will be thrown.

```
function staticWorkspaceErrors
    function nest
        % This will error since var Y is not explicitly specified
        load MatFileWithVarY
    end
end
```

To avoid an error, instead specify the variable name as an input to the `load` function.

```
function noStaticWorkspaceErrors
    function nest
        % This will not error since variables 'x' and 'y' are specified
        load MatFileWithVarX x
        y = load('MatFileWithVarY', 'y');
```

```
    end  
end
```

Alternatively, assign the output from the `load` function to a structure array.

Assigning a Variable in the MATLAB Debugger in a Nested Function

While debugging, you cannot add a variable using the debug command prompt if you are stopped in a nested function. Assign the variable into the base workspace, which is not static.

```
K>> assignin('base','X',myvalue)
```

Assigning a Variable in an Anonymous Functions

Anonymous functions cannot contain variable assignments. When the anonymous function is called an error will be thrown.

```
% This will error since 'x' is being assigned inside  
% the anonymous function  
@()eval("x=2")
```

Rewrite the function in such a way that variable assignments are not required.

```
xEquals2 = @()2;  
x = xEquals2()
```

```
x =
```

```
2
```

See Also

More About

- “Base and Function Workspaces” on page 20-9
- “Nested Functions” on page 20-27

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 and scripts in the folder immediately above the `private` subfolder.

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 20-37
- “Types of Functions” on page 20-17

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 Function or class whose name matches an explicitly imported name

The `import` function allows functions with compound names (names comprised of several parts joined by dots) to be called using only the final part of the compound name. When a function name matches an explicit (non-wildcard) imported function, MATLAB uses the imported compound name and gives it precedence over all other functions with the same name.

3 Nested functions within the current function

4 Local functions within the current file

5 Function or class whose name matches a wildcard-based imported name

When a function name matches a wildcard-based imported function, MATLAB uses the imported compound name and gives it precedence over all other functions with the same name, except for nested and local functions.

6 Private functions

Private functions are functions in a subfolder named `private` that is immediately below the folder of the currently running file.

7 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.

8 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.

9 Loaded Simulink® models

10 Functions in the current folder

11 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 Stateflow® chart with a .sfx extension
- 5 App file (.mlapp) created using MATLAB App Designer
- 6 Program file with a .mlx extension
- 7 P-file (that is, an encoded program file with a .p extension)
- 8 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.

Change in Rules For Function Precedence Order

Starting in R2019b, MATLAB changes the rules for name resolution, impacting the precedence order of variables, nested functions, local functions, and external functions. For information about the changes and tips for updating your code, see “Update Code for R2019b Changes to Function Precedence Order” on page 20-40.

- Identifiers cannot be used for two purposes inside a function
- Identifiers without explicit declarations might not be treated as variables
- Variables cannot be implicitly shared between parent and nested functions
- Change in precedence of compound name resolution
- Anonymous functions can include resolved and unresolved identifiers

The behavior of the `import` function has changed.

- Change in precedence of wildcard-based imports
- Fully qualified import functions cannot have the same name as nested functions
- Fully qualified imports shadow outer scope definitions of the same name
- Error handling when import not found
- Nested functions inherit import statements from parent functions

See Also

`import`

More About

- “What Is the MATLAB Search Path?”

- Variables on page 1-4
- “Types of Functions” on page 20-17
- “Class Precedence and MATLAB Path”

Update Code for R2019b Changes to Function Precedence Order

Starting in R2019b, MATLAB changes the rules for name resolution, impacting the precedence order of variables, nested functions, local functions, and external functions. The new rules simplify and standardize name resolution. For more information, see “Function Precedence Order” on page 20-37.

These changes impact the behavior of the `import` function. You should analyze and possibly update your code. To start, search your code for import statements. For example, use “Find Files and Folders” to search for `.m` and `.mlx` files containing text `import`. Refer to these search results when evaluating the effects of the following changes.

Identifiers cannot be used for two purposes inside a function

Starting in R2019b, an error results if you use an identifier, first as a local or imported function, and then as a variable. In previous releases, an identifier could be used for different purposes within the scope of a function, which resulted in ambiguous code.

If this behavior change impacts your code, rename either the variable or the function so that they have different names.

Starting in R2019b	Updated Code	R2019a and Earlier
The name <code>local</code> is used as the <code>local</code> function and then a variable. This code errors. <pre>function myfunc % local is an undefined variable local(1); % Errors local = 2; disp(local); end function local(x) disp(x) end</pre>	Rename the function <code>local</code> to <code>localFcn</code> . <pre>function myfunc localFcn(1); local = 2; disp(local); end function localFcn(x) disp(x) end</pre>	This code displays 1 then 2. <pre>function myfunc local(1); % local is a function local = 2; disp(local); end function local(x) disp(x) end</pre>

Identifiers without explicit declarations might not be treated as variables

Starting in R2019b, MATLAB does not use indexing operators to identify the variables in your program. Previously, an identifier without an explicit declaration was treated as a variable when it was indexed with a colon, end, or curly braces. For example, `x` was treated as a variable in `x(a,b,:)`, `x(end)`, and `x{a}`.

Consider the following code. MATLAB used to treat `x` as a variable because of colon-indexing. Starting in R2019b, if a function of the same name exists on the path, MATLAB treats `x` as a function.

```
function myfunc
load data.mat; % data.mat contains variable x
disp(x(:))
end
```

If you intend to use `x` as a variable from `data.mat` instead of a function, explicitly declare it. Similarly, to use an identifier `x` as a variable obtained from a script, declare it before invoking the script. This new behavior also applies if the variable is implicitly introduced by the functions `sim`, `eval`, `evalc`, and `assignin`.

This table shows some examples of how you can update your code.

Before	After
<pre>function myfunc load data.mat; disp(x(:)) end</pre>	<pre>function myfunc load data.mat x; disp(x(:)) end</pre>
<pre>function myfunc2 myscript; % Contains variable x disp(x(:)) end</pre>	<pre>function myfunc2 x = []; myscript; disp(x(:)) end</pre>

Variables cannot be implicitly shared between parent and nested functions

Starting in R2019b, sharing an identifier as a variable between a nested function on page 20-28 and its parent function is possible only if the identifier is explicitly declared as a variable in the parent function.

For example, in the following code, identifier `x` in `myfunc` is different from variable `x` in the nested function. If `x` is a function on the path, MATLAB treats `x` in `myfunc` as a function and the code runs. Otherwise, MATLAB throws an error.

```
function myfunc
nested;
x(3) % x is not a shared variable
    function nested
        x = [1 2 3];
    end
end
```

In previous releases, if `x` was a function on the path, MATLAB treated it as a function in `myfunc` and as a variable in `nested`. If `x` was not a function on the path, MATLAB treated it as a variable shared between `myfunc` and `nested`. This resulted in code whose output was dependent on the state of the path.

To use an identifier as a variable shared between parent and nested functions, you might need to update your code. For example, you can initialize the identifier to an empty array in the parent function.

Before	After
<pre>function myfunc nested; x(3) function nested x = [1 2 3]; end end</pre>	<pre>function myfunc x = []; nested; x(3) function nested x = [1 2 3]; end end</pre>

Change in precedence of wildcard-based imports

Starting in R2019b, imported functions from wildcard-based imports have lower precedence than variables, nested functions, and local functions. In R2019a and earlier, imports in a function shadowed local functions and nested functions.

For example, in this code, the statement `local()` calls `myfunc/local` instead of `pkg1.local` in the wildcard-based import. The statement `nest()` calls `myfunc/nest` instead of `pkg1.nest`.

Starting in R2019b	R2019a and Earlier
<pre>function myfunc % Import includes functions local and nest import pkg1.* local() % Calls myfunc/local function nest end nest(); % Calls myfunc/nest end function local end</pre>	<pre>function myfunc % Import includes functions local and nest import pkg1.* local() % Calls pkg1.local and % displays warning since R2018a function nest end nest(); % Calls pkg1.nest end function local end</pre>

In the search results for `import`, look for statements that include the wildcard character (*).

Fully qualified import functions cannot have the same name as nested functions

Starting in R2019b, fully qualified imports that share a name with a nested function in the same scope throw an error.

Starting in R2019b	Updated Code	R2019a and Earlier
<p>This function errors because it shares a name with a nested function in the same scope.</p> <pre>function myfunc import pkg.nest % Errors nest(); function nest end end</pre>	<p>To call function <code>nest</code> from the <code>import</code> statement, rename local function <code>myfunc/nest</code>.</p> <pre>function myfunc import pkg.nest nest(); function newNest end end</pre>	<p>This function calls function <code>nest</code> from the <code>import</code> statement.</p> <pre>function myfunc import pkg.nest nest(); % Calls pkg.nest function nest end end</pre>

Starting in R2019b	Updated Code	R2019a and Earlier
This function errors because declaring a variable with the same name as the imported function <code>nest</code> is not supported. <pre>function myvarfunc import pkg.nest % Errors nest = 1 end</pre>	Rename variable <code>nest</code> . <pre>function myvarfunc import pkg.nest % Errors thisNest = 1 end</pre>	This function modifies variable <code>nest</code> . <pre>function myvarfunc import pkg.nest nest = 1 % Modifies variable nest and % displays warning since R2019b end</pre>

Fully qualified imports shadow outer scope definitions of the same name

Starting in R2019b, fully qualified imports always shadow outer scope definitions of the same name. In R2019a and earlier, a fully qualified import was ignored when it shadowed an identifier in the outer scope.

Starting in R2019b	Updated Code	R2019a and Earlier
Local function <code>nest</code> calls function <code>x</code> from imported package. <pre>function myfunc x = 1; function nest % Import function x import pkg1.x % Calls pkg1.x x() end end</pre>	To use variable <code>x</code> in local function <code>nest</code> , pass the variable as an argument. <pre>function myfunc x = 1; nest(x) function nest(x1) % Import function x import pkg1.x % Calls pkg1.x with % variable x1 x(x1) end end</pre>	In this code, function <code>nest</code> ignores imported function <code>x</code> . <pre>function myfunc x = 1; function nest % Import function x import pkg1.x % x is a variable x() end end</pre>

Error handling when import not found

Starting in R2019b, fully qualified imports that cannot be resolved throw an error with or without Java. In R2019a and earlier, MATLAB behaved differently depending on whether you started MATLAB with the `-nojvm` option. Do not use functions like `javachk` and `usejava` to customize error messages.

Starting in R2019b	Updated Code	R2019a and Earlier
This code throws an error when starting MATLAB with the -nojvm option. <pre>function myfunc import java.lang.String % Errors % Do something with Java String class end if ~usejava('jvm') % Statement never executes disp('This function requires Java'); else % Do something with Java String class end end</pre>	Remove call to usejava. <pre>function myfunc import java.lang.String % Errors % Do something with Java String class end</pre>	This code displays a message when starting MATLAB with the -nojvm option. <pre>function myfunc import java.lang.String if ~usejava('jvm') % Display message disp('This function requires Java') else % Do something with Java String class end end</pre>

Nested functions inherit import statements from parent functions

Starting in R2019b, nested functions inherit `import` statements from the parent function. In R2019a and earlier, nested functions did not inherit import statements from their parent functions.

Starting in R2019b	R2019a and Earlier
<pre>function myfunc % Package p1 has functions plot and bar import p1.plot import p1.* nest function nest plot % Calls p1.plot bar % Calls p1.bar end end</pre>	<pre>function myfunc % Package p1 has functions plot and bar import p1.plot import p1.* nest function nest plot % Calls plot function on path bar % Calls bar function on path end end</pre>

Change in precedence of compound name resolution

Starting in R2019b, MATLAB resolves compound names differently. A compound name is comprised of several parts joined by a dot (for example, `a.b.c`), which can be used to reference package members. With R2019b, MATLAB resolves compound names by giving precedence to the longest matching prefix. In previous releases, the precedence order followed a more complex set of rules.

For example, suppose a package `pkg` contains a class `foo` with a static method `bar` and also a subpackage `foo` with a function `bar`.

```
+pkg/@foo/bar.m % bar is a static method of class foo
+pkg/+foo/bar.m % bar is a function in subpackage foo
```

In R2019b, a call to `which pkg.foo.bar` returns the path to the package function.

```
which pkg.foo.bar
+pkg/+foo/bar.m
```

Previously, a static method took precedence over a package function in cases where a package and a class had the same name.

Anonymous functions can include resolved and unresolved identifiers

Starting in R2019b, anonymous functions can include both resolved and unresolved identifiers. In previous releases, if any identifiers in an anonymous function were not resolved at creation time, all identifiers in that anonymous function were unresolved.

Starting in R2019b	R2019a and Earlier
<p>To evaluate the anonymous function, MATLAB calls the local function <code>lf</code> with <code>x</code> defined in <code>myscript</code> because <code>lf</code> in the anonymous function resolves to the local function.</p> <pre>function myfun myscript; % Includes x = 1 and lf = 10 f = @() lf(x); f() % Displays 'Inside lf' end % Local function to myfun function lf(y) disp('Inside lf'); end</pre>	<p>MATLAB considers <code>lf</code> as an unresolved identifier along with <code>x</code>, and used <code>x</code> to index into the variable <code>lf</code> from <code>myscript</code>.</p> <pre>function myfun myscript; % Includes x = 1 and lf = 10 f = @(); lf(x); f() % Displays 10 end % Local function to myfun function lf(y) disp('Inside lf'); end</pre>

See Also

[import](#)

More About

- “Import Classes”

Indexing into Function Call Results

This topic describes how to dot index into temporary variables created by function calls. A temporary variable is created when the result of a function call is used as an intermediate variable in a larger expression. The result of the function call in the expression is temporary because the variable it creates exists only briefly, and is not stored in the MATLAB workspace after execution. An example is the expression `myFunction(x).prop`, which calls `myFunction` with the argument `x`, and then returns the `prop` property of the result. You can invoke any type of function (anonymous, local, nested, or private) in this way.

Example

Consider the function:

```
function y = myStruct(x)
    y = struct("Afield",x);
end
```

This function creates a structure with one field, named `Afield`, and assigns a value to the field. You can invoke the function and create a structure with a field containing the value 1 with the command:

```
myStruct(1)
ans =
struct with fields:
    Afield: 1
```

However, if you want to return the field value directly, you can index into the function call result with the command:

```
myStruct(1).Afield
ans =
1
```

After this command executes, the temporary structure created by the command `myStruct(1)` no longer exists, and MATLAB returns only the field value. Conceptually, this usage is the same as creating the structure, indexing into it, and then deleting it:

```
S = struct("Afield",1);
S.Afield
clear S
```

Supported Syntaxes

MATLAB supports dot indexing into function call results, as in `foo(arg).prop`. Other forms of indexing into function call results (with parentheses such as `foo(arg)(2)` or with curly braces such as `foo(arg){2}`) are not supported. Successful commands must meet the criteria:

- The function is invoked with parentheses, as in `foo(arg1,arg2,...)`.
- The function returns a variable for which dot indexing is defined, such as a structure, table, or object.

- The dot indexing subscript is valid.

MATLAB always attempts to apply the dot indexing operation to the temporary variable, even if the function returns a variable for which dot indexing is not defined. For example, if you try to index into the matrix created by `magic(3)`, then you get an error.

```
magic(3).field  
Dot indexing is not supported for variables of this type.
```

You can add more indexing commands onto the end of an expression as long as the temporary variables can continue to be indexed. For example, consider the expression:

```
table(rand(10,2)).Var1(3,:)
```

In this expression, you index into a table to get the matrix it contains, and then index into the matrix to get the third row:

- `table(rand(10,2))` creates a table with one variable named `Var1`. The variable contains a 10-by-2 matrix.
- `table(rand(10,2)).Var1` returns the 10-by-2 matrix contained in `Var1`.
- `table(rand(10,2)).Var1(3,:)` returns the third row in the matrix contained in `Var1`.

See Also

[function](#) | [subsref](#)

More About

- “Types of Functions” on page 20-17
- “Array Indexing”
- “Access Data in Tables” on page 9-34

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 21-5
- “Validate Number of Function Arguments” on page 21-6
- “Checking Number of Arguments in Nested Functions” on page 21-8
- “Ignore Inputs in Function Definitions” on page 21-10
- “Check Function Inputs with validateattributes” on page 21-11
- “Parse Function Inputs” on page 21-13
- “Input Parser Validation Functions” on page 21-17

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 nargout > 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

- “Checking Number of Arguments in Nested Functions” on page 21-8
- “Comma-Separated Lists” on page 2-79

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

- “Checking Number of Arguments in Nested Functions” on page 21-8

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 21-5

Checking Number of Arguments 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 Inputs in Function Definitions

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.

In a file named `colorButton.m`, define a callback for a push button that does not use the `eventdata` input. Add a tilde to the input argument list so that the function ignores `eventdata`.

```
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 effectively the same as the following:

```
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 inputs in your function definition, in any position in the argument list. Separate consecutive tildes with a comma. For example:

```
function myFunction(myInput,~,~,~)
```

See Also

More About

- “Ignore Function Outputs” on page 1-3

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 name-value arguments, 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'

```

When using name-value arguments before R2021a, pass names as strings or character vectors, and separate names and values with commas. For example:

```
printPhoto('myfile.jpg','height',10,'width',8)
```

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 21-17

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 at (@) symbol.

Validation functions must accept a single input argument, and they must either return a scalar logical value (`true` or `false`) or error. If the validation function returns `false`, the Input Parser issues an error and your function stops processing.

There are several ways to define validation functions:

- Use an existing MATLAB function such as `ischar` or `isnumeric`. For example, check that a required input named `num` is numeric:

```
p = inputParser;
checknum = @isnumeric;
addRequired(p, 'num', checknum)

parse(p, 'text')
```

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
```

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 21-13
- “Create Function Handle” on page 13-2

More About

- “Anonymous Functions” on page 20-20

Debugging MATLAB Code

- “Debug a MATLAB Program” on page 22-2
- “Set Breakpoints” on page 22-7
- “Examine Values While Debugging” on page 22-12

Debug a MATLAB Program

You can debug your MATLAB program interactively in the Editor or programmatically using debugging functions in the Command Window. Both methods are interchangeable. To debug a program in the Live Editor or in the Editor in MATLAB Online, see “Debug Code in the Live Editor” on page 19-15.

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. You also can use the **F12** key to set the breakpoint.

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
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 22-7.

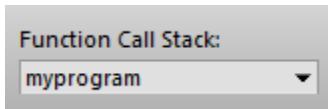
Run File

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 22-12

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

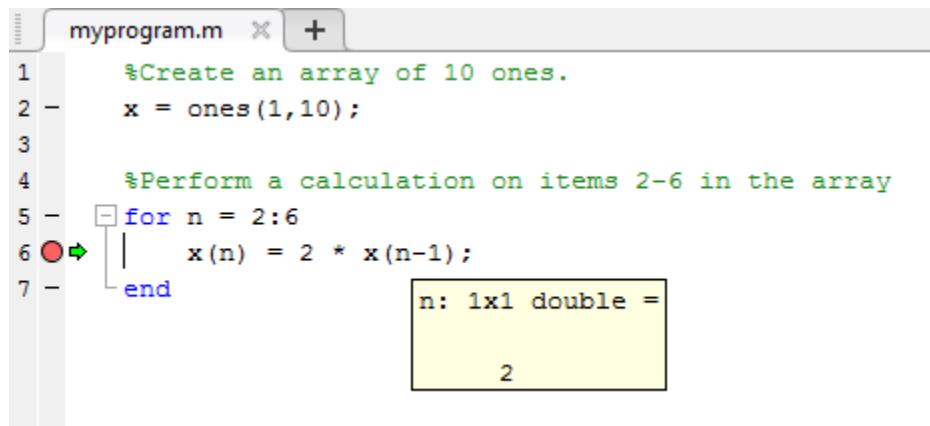
View the value of a variable while debugging to see whether a line of code has produced the expected result or not. To do this, position your mouse pointer to the left of the variable. The current value of the variable appears in a data tip.

```
for n = 2:6
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 22-12.

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`:



- 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

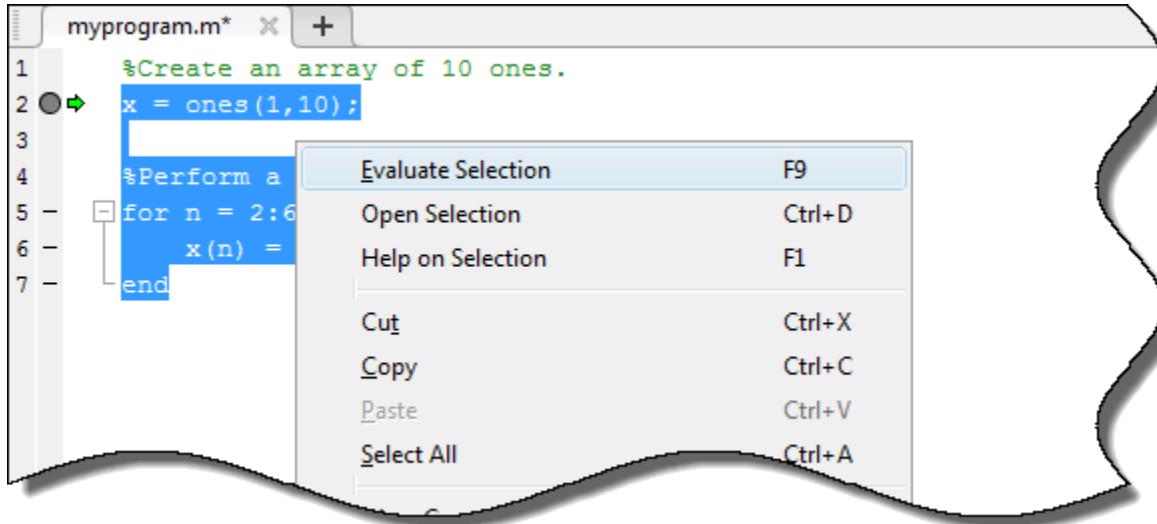
You can modify a section of code while debugging to test possible fixes without having to save your changes. Usually, it is a good practice to modify a MATLAB file after you quit debugging, and then save the modification and run the file. Otherwise, you might get unexpected results. However, there are 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.

- 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
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

Description	Toolbar Button	Function Alternative
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 22-7
- “Examine Values While Debugging” on page 22-12

Set Breakpoints

In this section...

- “Standard Breakpoints” on page 22-7
- “Conditional Breakpoints” on page 22-8
- “Error Breakpoints” on page 22-9
- “Breakpoints in Anonymous Functions” on page 22-10
- “Invalid Breakpoints” on page 22-10
- “Disable Breakpoints” on page 22-10
- “Clear Breakpoints” on page 22-11

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. You also can use the **F12** key to set the breakpoint.

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

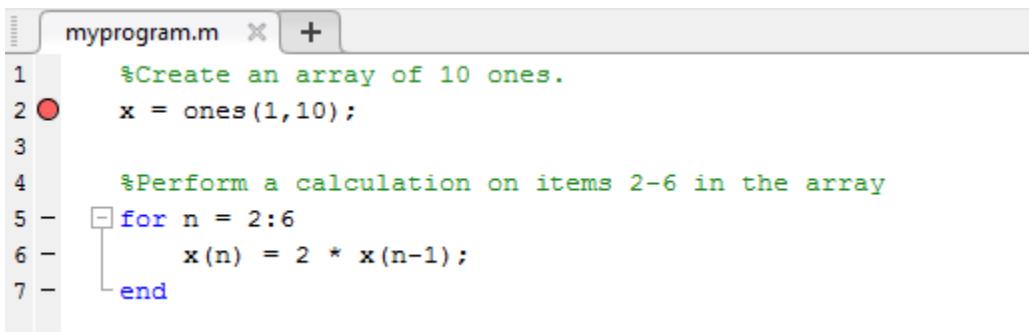
```

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.

To set a standard breakpoint programmatically, use 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`.

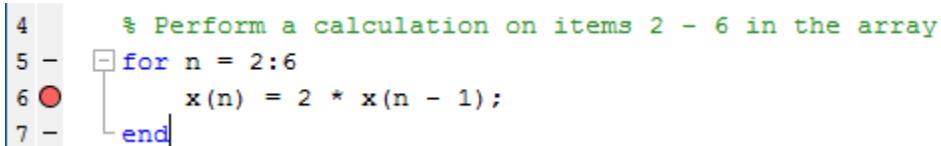


```

myprogram.m × +
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

```

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

```

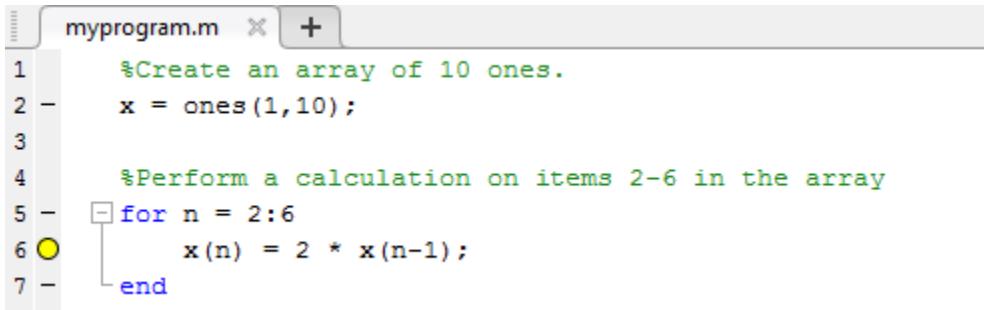
Conditional Breakpoints

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`.



The screenshot shows the MATLAB Editor window with a file named 'myprogram.m'. The code contains a for loop that iterates from 2 to 6. A conditional breakpoint is set on line 6, indicated by a yellow circle icon in the breakpoint alley. 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

```

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 ● sgr = @(x) x.^2;
3 - ⌘ sgr(5);
4 -

```

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:

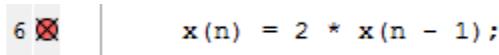
- There are unsaved changes in the file. To make breakpoints valid, save the file. The gray breakpoints become red, indicating that they are now valid.
- There is a syntax error in the file. When you set a breakpoint, an error message appears indicating where the syntax error is. To make the breakpoint 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 **F12** key to clear the breakpoint.

To clear a breakpoint programmatically, 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 22-12

Examine Values While Debugging

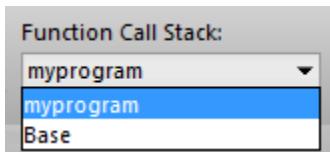
While your program is paused, you can view the value of any variable currently in the workspace. Examine values when you want to see whether a line of code produces the expected result or not. If the result is as expected, continue running or step to the next line. If the result is not as you expect, 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.

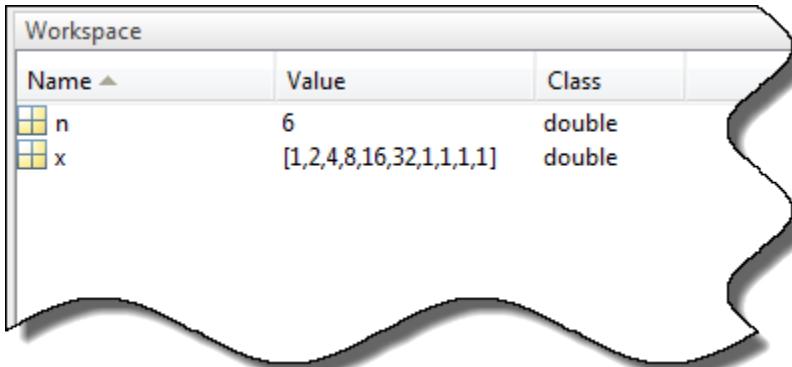
To list the variables in the current workspace, use `who` or `whos`.

View Variable Value

There are several ways to view the value of a variable while debugging a program:

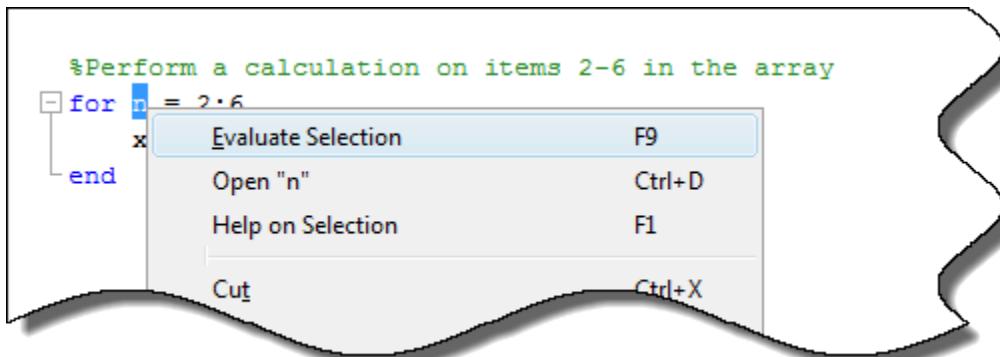
- View variable values in the Workspace browser and Variables Editor.

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.

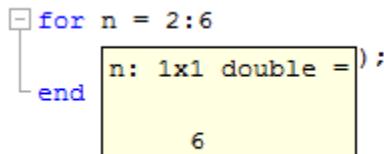
Use your mouse to select the variable or equation. Right-click and select **Evaluate Selection** from the context menu. The Command Window displays the value of the variable or equation.



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.



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.

- On the **Home** tab, in the **Environment** section, click **Preferences**. Then select **MATLAB** > **Editor/Debugger** > **Display**.
 - 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 22-7

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.

- “Publish and Share MATLAB Code” on page 23-2
- “Publishing Markup” on page 23-6
- “Output Preferences for Publishing” on page 23-21

Publish and Share MATLAB Code

MATLAB provides options for presenting your code to others. You can publish your MATLAB Code files (.m) to create formatted documents or you can create and share live scripts and live functions in the Live Editor.

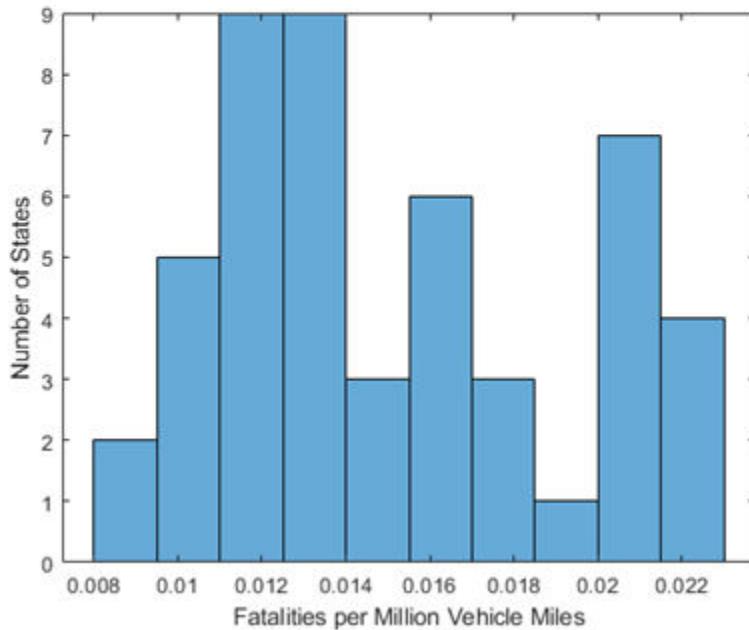
Create and Share Live Scripts in the Live Editor

The easiest way to create cohesive, sharable documents that include executable MATLAB code, embedded output, and formatted text is to use the Live Editor. Supported output formats include: MLX, PDF, Microsoft Word, HTML, and LaTeX. For details, see “Create Live Scripts in the Live Editor” on page 19-6.

Distribution of Fatalities

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')
```



Publish MATLAB Code Files (.m)

To create shareable documents using your MATLAB Code files (.m), you can publish the files. Publishing a MATLAB Code file 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.

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 %> Note About Gibbs Phenomenon %> Even though the approximations are constantly %> improving, they will never be exact because of the %> Gibbs phenomenon, or ringing.</pre>	<h3>Square Waves from Sine Waves</h3> <p>The Fourier series expansion for a square-wave is made up of a sum of odd harmonics, as shown here using MATLAB®.</p> <h4>Contents</h4> <ul style="list-style-type: none"> Add an Odd Harmonic and Plot It Note About Gibbs Phenomenon <h4>Add an Odd Harmonic and Plot It</h4> <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	<code>%% Square Waves from Sine Waves</code> <code>%% Add an Odd Harmonic and Plot It</code> <code>%% Note About Gibbs Phenomenon</code>
Variable name in italics	<code>% As <u>k</u> increases, ...</code>
LaTeX equation	<code>% \$\$ y = y + \frac{\sin(k*t)}{k} \$\$</code>

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 23-22.

In MATLAB Online, to allow MATLAB to open output windows automatically when publishing, enable pop-up windows in your Web browser.

After publishing the code, you can share the folder containing the published files. For more information, see “Share Folders in MATLAB”. In MATLAB Online, you also can make the results publicly available by copying the published files from the `html` folder to the `Published` folder. Then, you can use a URL of the form `https://matlab.mathworks.com/users/userid/Published/filename/index.html` (for HTML) or `https://matlab.mathworks.com/users/userid/Published/foldername/filename.pdf` (for PDF) to share the files.

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'))
```

Add Help and Create Documentation

You can add help to your code by inserting comments at the start of a MATLAB code file. MATLAB displays the help comments when you type `help file_name` in the Command Window. For more information, see “Add Help for Your Program” on page 20-5.

You also can create your own MATLAB documentation topics for viewing from the MATLAB Help browser or the web. For more information, see “Display Custom Documentation” on page 31-21

See Also

`publish`

More About

- “Create Live Scripts in the Live Editor” on page 19-6
- “Publishing Markup” on page 23-6
- “Output Preferences for Publishing” on page 23-21

External Websites

- Publishing MATLAB Code from the Editor video

Publishing Markup

In this section...

- “Markup Overview” on page 23-6
- “Sections and Section Titles” on page 23-8
- “Text Formatting” on page 23-9
- “Bulleted and Numbered Lists” on page 23-10
- “Text and Code Blocks” on page 23-10
- “External File Content” on page 23-11
- “External Graphics” on page 23-12
- “Image Snapshot” on page 23-14
- “LaTeX Equations” on page 23-14
- “Hyperlinks” on page 23-16
- “HTML Markup” on page 23-18
- “LaTeX Markup” on page 23-19

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 23-8	<pre>%% SECTION TITLE % DESCRIPTIVE TEXT %%% SECTION TITLE WITHOUT SECTION BREAK % DESCRIPTIVE TEXT</pre>

Result in Output	Example of Corresponding File Markup
“Text Formatting” on page 23-9	<pre>% _ITALIC TEXT_ % *BOLD TEXT* % MONOSPACED TEXT </pre> <p style="text-align: center;">% Trademarks: % TEXT(TM) % TEXT(R)</p>
“Bulleted and Numbered Lists” on page 23-10	<pre>%% Bulleted List % % * BULLETED ITEM 1 % * BULLETED ITEM 2 % %% Numbered List % % # NUMBERED ITEM 1 % # NUMBERED ITEM 2 %</pre>
“Text and Code Blocks” on page 23-10	<pre>%% % % PREFORMATTED % TEXT % %% MATLAB(R) Code % % for i = 1:10 % disp x % end %</pre>
“External File Content” on page 23-11	<pre>%<include>filename.m</include></pre>
“External Graphics” on page 23-12	<pre>%<><FILENAMEN.PNG>></pre>
“Image Snapshot” on page 23-14	<pre>snapnow;</pre>
“LaTeX Equations” on page 23-14	<pre>%% Inline Expression % \$x^2+e^{\pi i}\$ %% Block Equation % % \$\$e^{\pi i} + 1 = 0\$\$</pre>
“Hyperlinks” on page 23-16	<pre>% <https://www.mathworks.com MathWorks> % <matlab:FUNCTION DISPLAYED_TEXT></pre>

Result in Output	Example of Corresponding File Markup
“HTML Markup” on page 23-18	<pre>%<html> % <table border=1><tr> % <td>one</td> % <td>two</td></tr></table> % </html> %</pre>
“LaTeX Markup” on page 23-19	<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.

Basic Matrix Operations in MATLAB®

This is a demonstration of some aspects of MATLAB® and the Symbolic Math Toolbox™.

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');
```

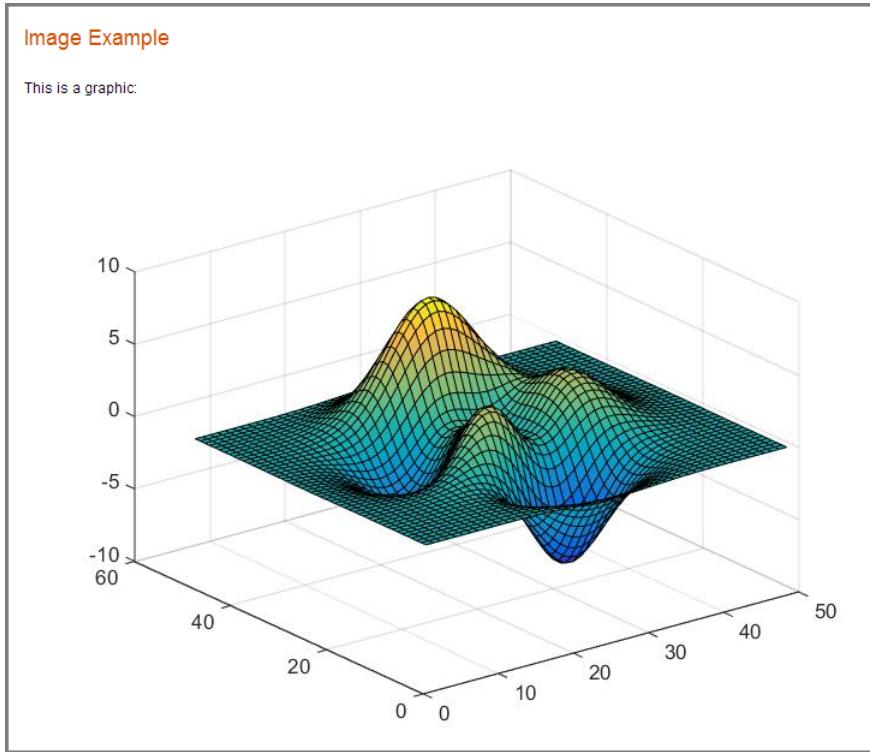
To produce an HTML file containing surfpeaks.jpg from a MATLAB file:

- 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.

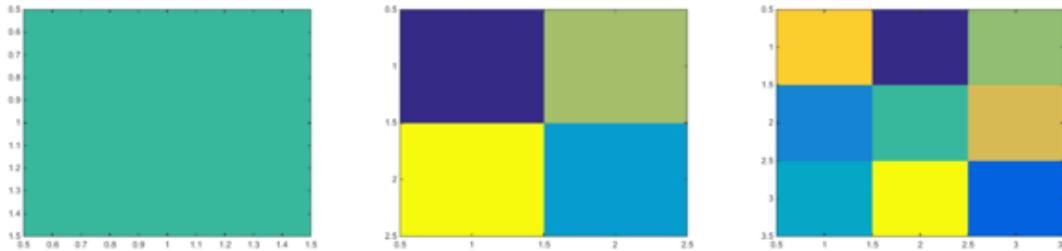
```
%% Scale magic Data and Display as Image

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 23-21.

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 Inline Expression Example

This is an equation: $x^2 + e^{\pi i}$. It is inline with the text.

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:

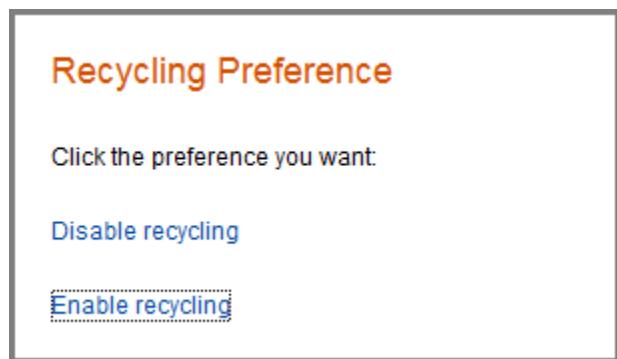
- “Dynamic Link to Run Code” on page 23-17
- “Dynamic Link to a File” on page 23-17
- “Dynamic Link to a MATLAB Function Reference Page” on page 23-17

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.



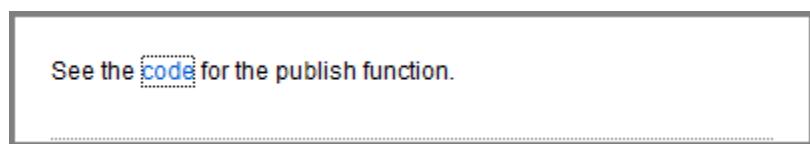
When you click one of the hyperlinks, MATLAB sets the `recycle` command accordingly. After clicking a hyperlink, run `recycle` in the Command Window to confirm that the setting is as you expect.

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>')
```



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
$ & $ \\ \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

- “Publish and Share MATLAB Code” on page 23-2
- “Output Preferences for Publishing” on page 23-21

Output Preferences for Publishing

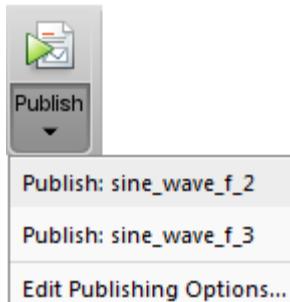
In this section...

- “How to Edit Publishing Options” on page 23-21
- “Specify Output File” on page 23-22
- “Run Code During Publishing” on page 23-22
- “Manipulate Graphics in Publishing Output” on page 23-24
- “Save a Publish Setting” on page 23-27
- “Manage a Publish Configuration” on page 23-28

How to Edit Publishing Options

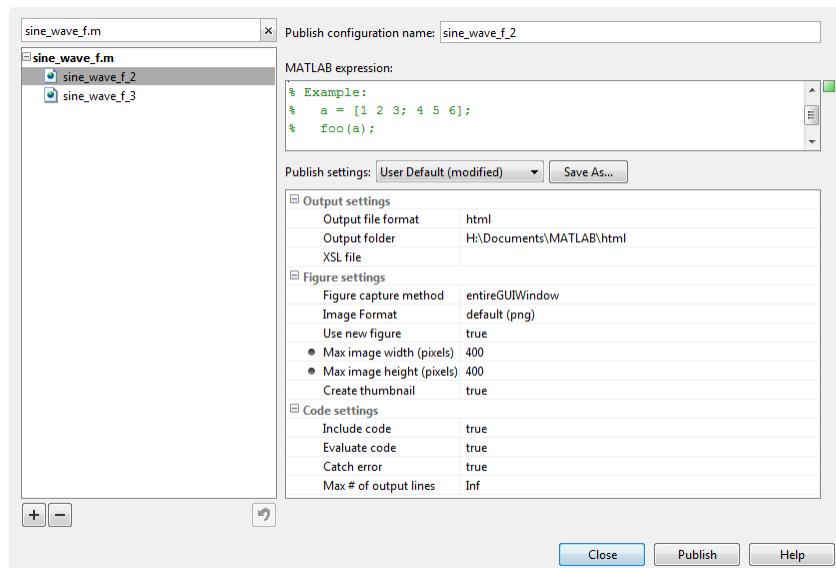
Use the default publishing preferences if your code requires no input arguments and you want to publish to HTML. However, if your code requires input arguments, or if you want to specify output settings, code execution, or figure formats, then specify a custom configuration.

- 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.
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 23-22
- “Evaluating Code” on page 23-23
- “Including Code” on page 23-23
- “Catching Errors” on page 23-23
- “Limiting the Amount of Output” on page 23-24

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.

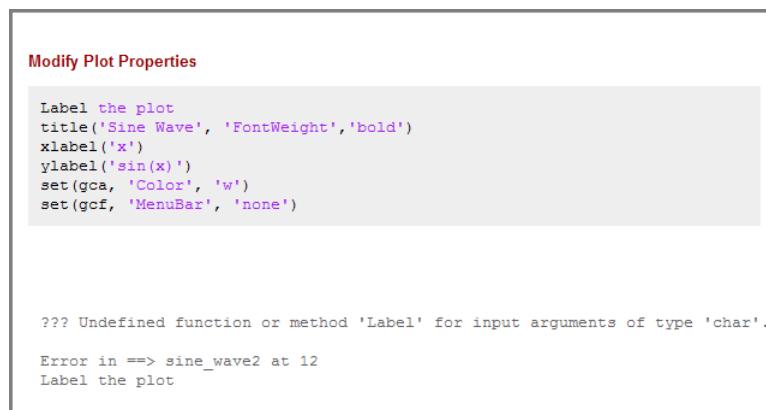
Note Publish configurations use the base MATLAB workspace. Therefore, a variable in the **MATLAB expression** pane overwrites the value for an existing variable in the base workspace.

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.



The screenshot shows the 'Modify Plot Properties' dialog box. In the code editor area, the following MATLAB code is displayed:

```
Label the plot
title('Sine Wave', 'FontWeight','bold')
xlabel('x')
ylabel('sin(x)')
set(gca, 'Color', 'w')
set(gcf, 'MenuBar', 'none')
```

Below the code, an error message is shown:

```
??? Undefined function or method 'Label' for input arguments of type 'char'.
Error in ==> sine_wave2 at 12
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 23-24
- “Setting an Image Size” on page 23-24
- “Capturing Figures” on page 23-25
- “Specifying a Custom Figure Window” on page 23-25
- “Creating a Thumbnail” on page 23-26

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 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.

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 23-12
- 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 23-12.

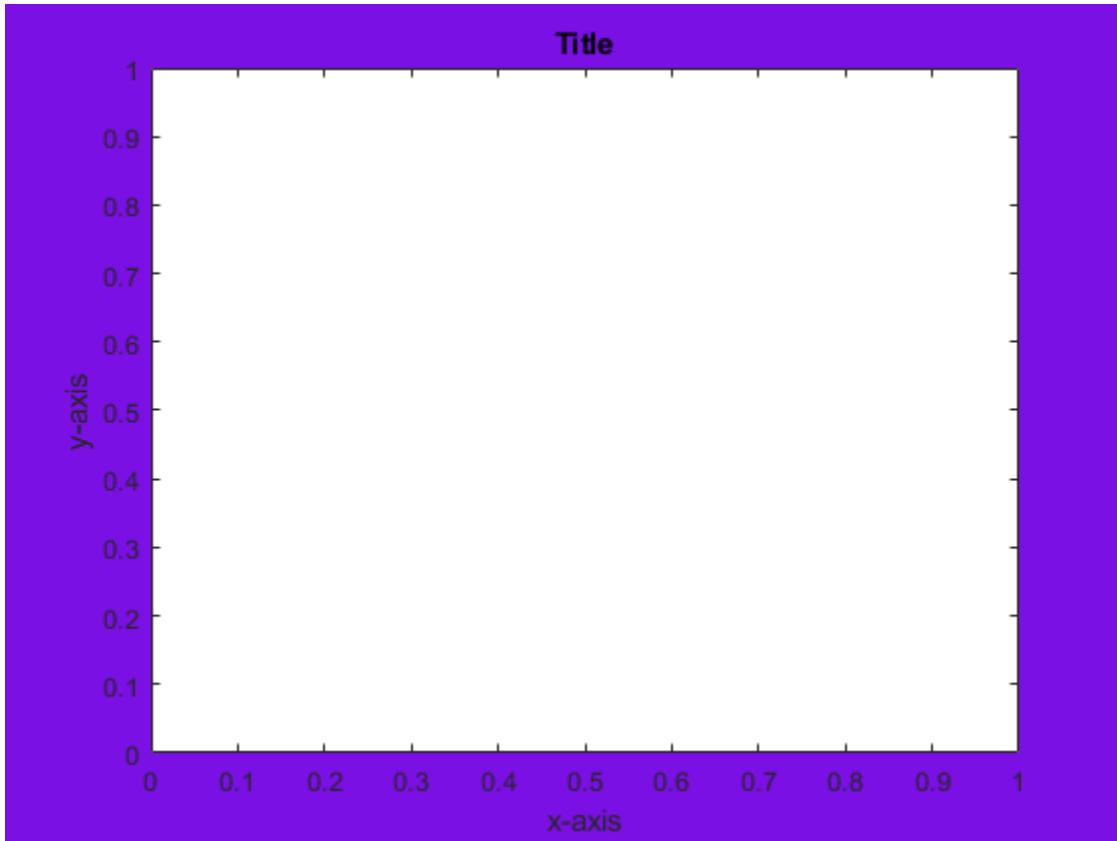
Use the following code as a template to produce Figure windows that meet your needs.

```
% 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

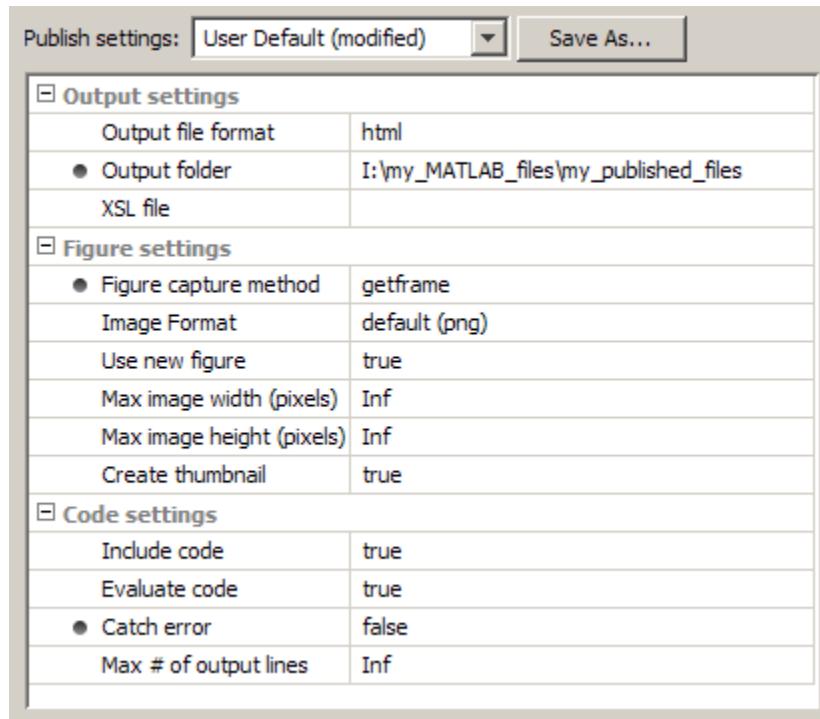
You can save the first code-generated graphic as a thumbnail image. You can use this thumbnail to represent your file on HTML pages. To create a thumbnail, follow these steps:

- 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 23-28
- “Creating Multiple Publish Configurations for a File” on page 23-28
- “Reassociating and Renaming Publish Configurations” on page 23-29
- “Using Publish Configurations Across Different Systems” on page 23-29

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  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  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.

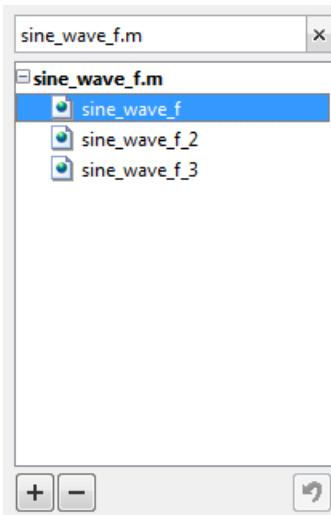
Creating Multiple Publish Configurations for a File

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

- “Publish and Share MATLAB Code” on page 23-2
- “Publishing Markup” on page 23-6

Coding and Productivity Tips

- “Save and Back Up Code” on page 24-2
- “Check Code for Errors and Warnings” on page 24-5
- “Improve Code Readability” on page 24-16
- “Find and Replace Text in Files” on page 24-22
- “Add Reminders to Files” on page 24-29
- “MATLAB Code Analyzer Report” on page 24-31
- “MATLAB Code Compatibility Report” on page 24-34

Save and Back Up Code

In this section...

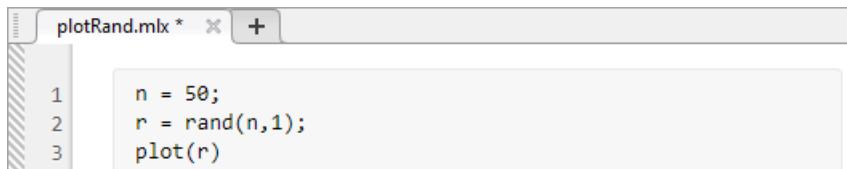
- “Save Code” on page 24-2
- “Back Up Code” on page 24-2
- “Recommendations on Saving Files” on page 24-3
- “File Encoding” on page 24-3

You can save your files in the Editor and Live Editor using several methods. In the Editor, you also can create backup copies of your files. Creating backup copies of your files ensures that you have a known working version of the files before making changes to them, and can also be useful for recovering lost changes after a system problem.

Depending on your needs, you can also control how the files you save are encoded and cached.

Save Code

When you modify a file in the Editor or the Live Editor, MATLAB indicates that there are unsaved changes in the file by displaying an asterisk (*) next to the file name in the document tab.



```
plotRand.mlx * +
```

```

1 n = 50;
2 r = rand(n,1);
3 plot(r)

```

To save the file, go to the **Editor** or **Live Editor** tab, and in the **File** section, click  **Save**.

To change the name, location, or type of a file, select **Save > Save As**. For example, to save a live script as a plain code file (.m), on the **Live Editor** tab, in the **File** section, select **Save > Save As**. In the dialog box that appears, select **MATLAB Code files (UTF-8) (*.m)** as the **Save as type** and click **Save**.

Back Up Code

You can create backup copies of your files in the Editor. Creating a backup copy of a file ensures that you have a known working version of the file before making changes to it. To create a backup copy of a file, on the **Editor** tab, in the **File** section, select **Save > Save Copy As**. This option is not available in the Live Editor or in MATLAB Online.

In addition, when you modify files in the Editor, MATLAB automatically creates backup copies of the files. If you lose changes to your files due to system problems, you can use the automatically created backup copies of the files to recover the changes.

By default, MATLAB saves a backup copy of a modified file every five minutes using the same file name but with an .asv extension. For example, `filename.m` would have a backup file name of `filename.asv`. If you lose changes to your file, you can recover the unsaved changes by opening the backup copy of the file, `filename.asv`, and saving it as `filename.m`.

To change how and when MATLAB saves backup copies of files, on the **Home** tab, in the **Environment** section, click  **Preferences**. Then, select **MATLAB > Editor/Debugger > Backup Files**. You can specify:

- How often to save backup copies of the files you are editing.
- What file extension to use when creating backup copies of files.
- Where to save backup copies of files.
- Whether to automatically delete backup copies of files when you close the corresponding source file in the Editor.

For more information about the available options, see the Backup Files preferences in “Editor/Debugger Preferences”.

In MATLAB Online, every time you save a code file in the Editor, MATLAB stores the contents of your code file in the version history. For more information about recovering a previous version of a file in MATLAB Online, see “Restore Files in MATLAB Online”.

MATLAB does not automatically create backups of files modified in the Live Editor.

Recommendations on Saving Files

MathWorks recommends that you save files that you create to a folder outside the *matlabroot* folder tree, where *matlabroot* is the folder returned when you type `matlabroot` in the Command Window. Similarly, when you edit files that you get from MathWorks, save your edited version outside the *matlabroot* folder tree. If you save your files in the *matlabroot* folder tree, they can be overwritten when you install a new version of MATLAB.

If you do save files in the *matlabroot* folder tree, you may need to take extra steps for your changes to take effect. 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 add, remove, or make changes to files in the *matlabroot* folder tree using an external editor or file system operations, you must update the cache so that MATLAB recognizes the changes you made. For more information, see “Toolbox Path Caching in MATLAB”.

File Encoding

As of R2020a, when the Editor saves a new MATLAB code file that has a `.m` extension, such as a script or a function, it uses UTF-8 without a byte-order-mark (BOM). The Editor saves existing files with their current encoding unless a different one is selected from the Save As dialog. For example, to save a file using a legacy locale-specific encoding for compatibility with an earlier release of MATLAB, on the **Editor** tab, in the **File** section, select **Save > Save as**. In the dialog box that appears, select the desired encoding from the **Save as type** options.

The current encoding is displayed next to the file name in the Editor status bar or, if the Editor Window is docked, the Desktop status bar.

See Also

More About

- “Editor/Debugger Preferences”
- “Manage Files and Folders”

Check Code for Errors and Warnings

MATLAB Code Analyzer can automatically check your code for coding problems.

Automatically Check Code in the Editor and Live 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 and Live 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**.

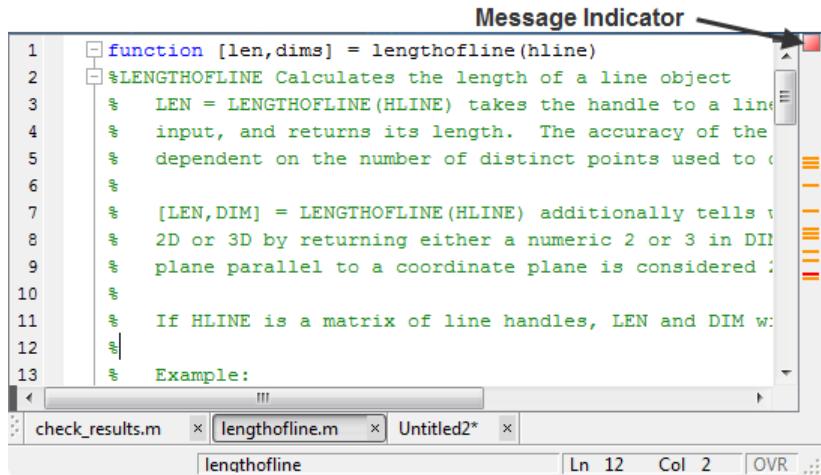
Use Continuous Code Checking

You can use continuous code checking in MATLAB code files in the Editor and Live Editor:

- 1** Open a MATLAB code file in the Editor or Live 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 or other significant issues were detected.
 - **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.



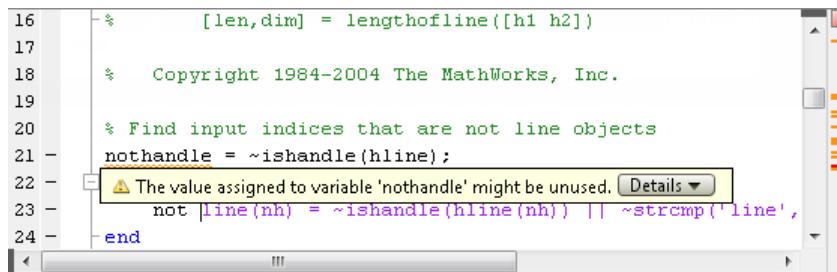
- 3 Click the message indicator to go to the next code fragment containing a message. The next code fragment is relative to the current cursor position, viewable in the status bar.

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.

- 4 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.



- 5 Click the **Details** button.

The window expands to display an explanation and user action.

- 6 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.

- 7 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.

A screenshot of the MATLAB Editor showing a code file. A warning message is displayed in a yellow box: "⚠ NUMEL(x) is usually faster than PROD(SIZE(x)). [Fix] compute the length". The code snippet is as follows:

```

25 len = zeros(size(hline));
26 -
27 - for nl = 1:prod(size(hline))
28 -     if ~notline(nl)
29 -         flds = get(hline(nl));
30 -         fdata = {'XData','YData','ZData'};
31 -         for nd = 1:length(fdata)
32 -             data(nd) = getfield(flds,fdata(nd));
33 -         end
34 -     end
35 - end

```

- 8** 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.

A screenshot of the MATLAB Editor showing a code file with three error messages in the message bar:

- Line 47: Invalid syntax at ')'. Possibly, a ')' is missing.
- Line 47: Invalid syntax at ')'. Possibly, a ')' is missing.
- Line 47: Parse error at ')': usage might be invalid MATLAB syntax.

The cursor is over the third message. The code snippet is as follows:

```

21 - nothandle = ~ishandle(hline);
22 - for nh = 1:prod(size(hline))
23 -     notline(nh) = ~ishandle(hline(nh)) || ~strcmp('line',
24 - end
25 -
26 - 
27 - 
28 - 
29 - 
30 -     flds = get(hline(nl));
31 -     ...

```

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 or Live 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.

Creating a report for an individual file in the Live Editor is not supported.

- 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 24-31.
 - 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 24-5. 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 48, 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 48, 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 24-9
- “Suppress All Instances of a Message in the Current File” on page 24-9
- “Suppress All Instances of a Message in All Files” on page 24-10
- “Save and Reuse Code Analyzer Message Settings” on page 24-10

You cannot suppress error messages such as syntax errors. Therefore, instructions on suppressing messages do not apply to those types of messages.

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 24-5 , follow these steps:

- 1 In line 48, 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 to suppress the **Terminate statement with semicolon to suppress output (in functions)** Code Analyzer message for 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 24-5, ignoring both messages for line 48 adds the comment `%#ok<NBRAK, NOPRT>` at the end of the line.

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 show the **Terminate statement with semicolon to suppress output (in functions)** Code Analyzer message for 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 24-5, follow these steps:

- 1 In line 48, 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 to suppress all instances of the **Terminate statement with semicolon to suppress output (in**

functions) Code Analyzer message in the current 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 24-5, ignoring both messages for line 48 adds the comment %#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 show all instances of the Terminate statement with semicolon to suppress output (in functions) Code Analyzer message in the current 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 24-5, follow these steps:

- 1 In line 48, right-click at the first underline (for a single-button mouse, press **Ctrl+click**).
- 2 Select **Suppress 'Terminate statement with semicolon...' > In All Files**.

This modifies the Code Analyzer preference setting.

If you know which message or messages that 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

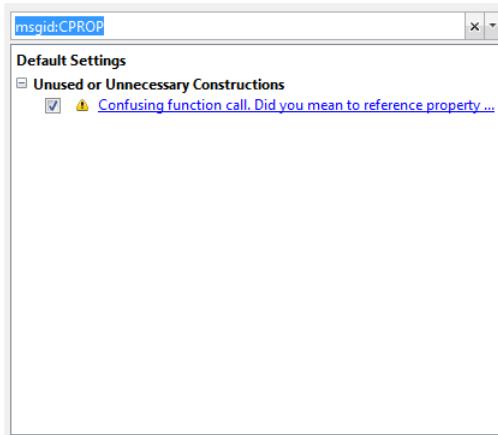
If you receive code that contains suppressed messages, you might want to review those messages without the need to unsuppress them first. A message might be in a suppressed state for any of the following reasons:

- 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:CPR0P 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 24-8.

These sections describe code analysis limitations regarding the following:

- “Distinguish Function Names from Variable Names” on page 24-13
- “Distinguish Structures from Handle Objects” on page 24-13
- “Distinguish Built-In Functions from Overloaded Functions” on page 24-14
- “Determine the Size or Shape of Variables” on page 24-14
- “Analyze Class Definitions with Superclasses” on page 24-14

- “Analyze Class Methods” on page 24-14

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 24-8.

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 24-10, includes the status of this setting.

Improve Code Readability

In this section...

"Indenting Code" on page 24-16

"Right-Side Text Limit Indicator" on page 24-17

"Code Folding — Expand and Collapse Code Constructs" on page 24-18

"Code Refactoring — Automatically convert selected code to a function" on page 24-21

Indenting Code

Indenting code makes reading statements such as `while` loops easier. To set and apply indenting preferences to code in the Editor and Live Editor:

- 1 On the **Home** tab, in the **Environment** section, click  **Preferences**.
- 2 The Preferences dialog box opens.
- 3 Select **MATLAB > Editor/Debugger > Language**.
- 4 Choose a computer language from the **Language** drop-down list.
- 5 In the **Indenting** section, select or clear **Apply smart indenting while typing**, depending on whether you want indenting applied automatically, as you type.

If you clear this option, you can manually apply indenting by selecting the lines in the Editor and Live 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 and Live Editor align the function code with the function declaration.
 - **Indent nested functions** — The Editor and Live Editor indent the function code within a nested function.
 - **Indent all functions** — The Editor and Live Editor indent 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.

In MATLAB Online, indenting preferences are located under **MATLAB > Editor/Debugger > MATLAB Language** and **MATLAB > Editor/Debugger > Other Languages**.

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** or **Live Editor** tab, click or .
- 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.

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, go to the **Home** tab and in the **Environment** section, click  **Preferences**. Select **MATLAB > Editor/Debugger > Language**, and adjust the **Comment formatting** preferences. To adjust **Comment formatting** preferences in MATLAB Online, select **Editor/Debugger > MATLAB Language**.

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, go to the **Home** tab, and in the **Environment** section, click  **Preferences**. Then, select **Editor/Debugger > Code Folding**.

To expand or collapse code, click the plus  or minus sign  that appears to the left of the construct in the Editor. You also use the **Ctrl+Shift+. (period)** and **Ctrl+. (period)** keyboard shortcuts, or use the code folding buttons in the **View** tab.

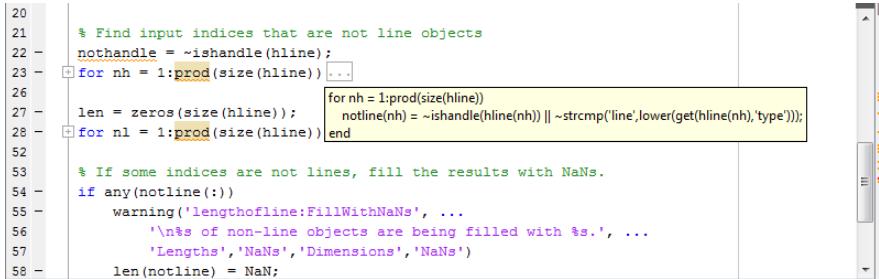
To expand or collapse all of the code in a file, place your cursor anywhere within the file, right-click, and then select **Code Folding > Expand All** or **Code Folding > Fold All** from the context menu. You also can use the **Ctrl+Shift+, (comma)** and **Ctrl+, (comma)** keyboard shortcuts, or the code folding buttons in the **View** tab.

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 . The code appears in a tooltip. Viewing folded code in a tooltip is not supported in MATLAB Online.

The following image shows the tooltip that appears when you place the pointer over the ellipsis on line 23 of `lengthofline.m` when a **for** loop is folded.



```

20
21 % Find input indices that are not line objects
22 - nothandle = ~ishandle(hline);
23 - for nh = 1:prod(size(hline)) ...
24 - len = zeros(size(hline));
25 - for nh = 1:prod(size(hline))
26 - len(nh) = -ishandle(hline(nh)) || ~strcmp('line',lower(get(hline(nh),'type')));
27 - end
28 - if any(notline())
29 - warning('lengthofline:FillWithNaNs', ...
30 - '\n% of non-line objects are being filled with %s.', ...
31 - 'Lengths','NaNs','Dimensions','NaNs')
32 - len(notline) = NaN;
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58

```

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. This is because MATLAB does not include trailing white space and comments in a function definition that has no explicit `end` statement. In MATLAB Online, the Editor does include the line when folding the function.

"Code Folding Enabled for Function Code Only" on page 24-20 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 24-20 and "Code Folding Enabled for Sections Only" on page 24-20 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 24-21, 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

Code Refactoring — Automatically convert selected code to a function

In the Live Editor, you can break large live scripts or functions into smaller pieces by automatically converting selected areas of code into functions or local functions. This is called code refactoring.

To refactor a selected area of code:

- 1 Select one or more lines of code.
- 2 On the **Live Editor** tab, in the **Code** section, click  **Refactor** and select from the available options.
- 3 Enter a name for the new function. MATLAB creates a function with the selected code and replaces the original code with a call to the newly created function.

Refactoring is also available in the Editor in MATLAB Online. To refactor a selected area of code in the MATLAB Online Editor, on the **Editor** tab, in the **Code** section, click the code refactoring  button. Then, select from the available options.

Find and Replace Text in Files

In this section...

- “Find and Replace Any Text in Current File” on page 24-22
- “Find and Replace Functions or Variables in Current File” on page 24-22
- “Automatically Rename All Functions or Variables in a File” on page 24-23
- “Find Text in Multiple File Names or Files” on page 24-24
- “Function Alternative for Finding Text” on page 24-25
- “Go To Location in File” on page 24-25

Find and Replace Any Text in Current File

You can search for, and optionally replace, any text within a file open in the Editor or Live Editor. To search for text in a file, on the **Editor** or **Live Editor** tab, in the **Navigate** section, click  **Find**. You also can use the **Ctrl+F** keyboard shortcut.

In the find and replace dialog box, enter the text that you want to search for and then use the **Find Next** and **Find Previous** buttons to search forward or backward through the file. In the Live Editor and in MATLAB Online, use the  and  buttons or the **F3** and **Shift+F3** keyboard shortcuts instead.

To find text that matches the case of the search text, select the **Match case** check box. To find an exact full-word match, select the **Whole word** check box. In the Live Editor and in MATLAB Online, these options are available using the  and  buttons.

To replace text in the file, enter the text that you want to replace the search text with and then use the **Replace** or **Replace all** buttons to replace the text. In the Live Editor and MATLAB Online, click the  expand button to the left of the search field to open the replace options. Then, enter the text that you want to replace the search text with and use the  and  buttons to replace the text.

Find and Replace Functions or Variables in 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 20-10, 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.

```

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

```

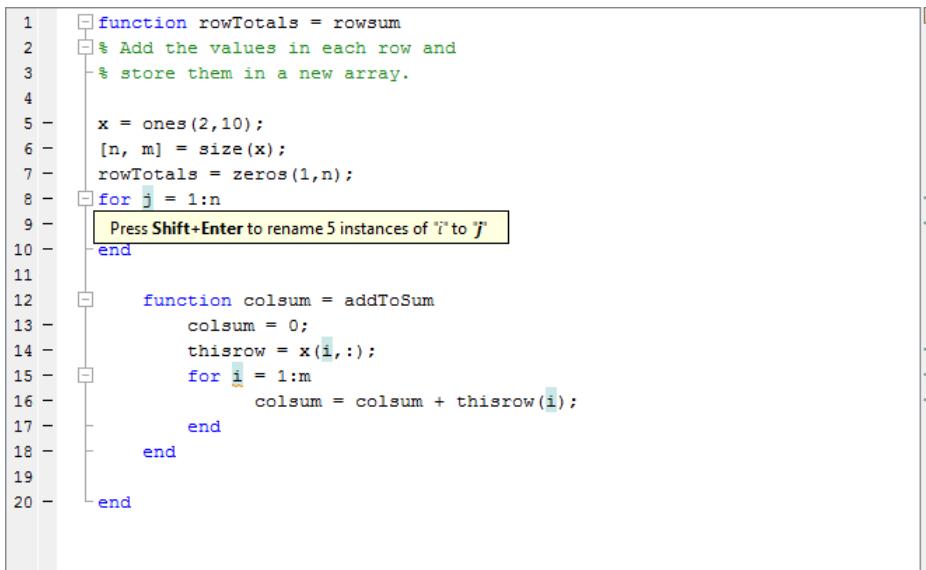
To disable automatic highlighting, go to the **Home** tab and in the **Environment** section, click **Preferences**. In **MATLAB > Colors > Programming Tools**, clear the **Automatically highlight** option.

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 <code>foo</code> in: <code>function foo(m)</code>
Input or output variable name in a function declaration	Rename <code>y</code> or <code>m</code> in: <code>function y = foo(m)</code>
Variable name on the left side of assignment statement	Rename <code>y</code> 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**.



The screenshot shows a portion of a MATLAB script in the Editor. Line 8 contains a for loop with the variable `i`. A tooltip box appears over the `i` in the `for i = 1:n` line, containing the text "Press Shift+Enter to rename 5 instances of 'i' to 'j'". The rest of the script includes nested functions `rowTotals` and `colsum`.

```
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     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**.

In MATLAB Online, variable and function renaming preferences are located under **MATLAB > Editor/Debugger > MATLAB Language**.

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. To open the Find Files dialog box in MATLAB Online, on the **Editor** or **Live Editor** tab, in

the **Navigate** section, click  **Find** and select **Find Files**. For more information, 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.

Go To Location in File

You can go to a specific location in a file, set bookmarks, navigate backward and forward within the file, and open a file or variable from within a file.

Navigate to a Specific Location

This table shows how to navigate to a specific location within a file open in the Editor and Live Editor.

Go To	Instructions	Notes
Line Number	On the Editor or Live Editor tab, in the Navigate section, click  Go To  . Select Go to Line... and specify the line that you want to navigate to.	None
Function definition	On the Editor or Live Editor tab, in the Navigate section, click  Go To  . In the Function section, select the local function or nested function that you want to navigate to. You also can select the file in the Current Folder browser and click the up arrow  at the bottom of Current Folder browser to open the Details panel. Then, in the Details panel, double-click the function icon  corresponding to the title of the function or local function that you want to navigate to.	Includes local functions and nested functions. 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.

Go To	Instructions	Notes
Code Section	<p>On the Editor or Live Editor tab, in the Navigate section, click Go To. In the Sections section, select the title of the code section that you want to navigate to.</p> <p>You also can select the file in the Current Folder browser and click the up arrow at the bottom of Current Folder browser to open the Details panel. Then, in the Details panel, double-click the section icon corresponding to the title of the section that you want to navigate to.</p>	For more information, see “Divide Your File into Code Sections” on page 18-5.
Property	<p>In the Current Folder browser, select the file that you want to navigate through and click the up arrow at the bottom of Current Folder browser to open the Details panel. Then, in the Details panel, double-click the property icon corresponding to the name of the property that you want to navigate to.</p>	For more information, see “Ways to Use Properties”.
Method	<p>In the Current Folder browser, select the file that you want to navigate through and click the up arrow at the bottom of Current Folder browser to open the Details panel. Then, in the Details panel, double-click the icon corresponding to the name of the method that you want to navigate to.</p>	For more information, see “Methods in Class Design”.
Bookmark	<p>On the Editor tab, in the Navigate section, click Go To. In the Bookmarks section, select Previous or Next.</p> <p>To navigate to a bookmark in the Live Editor and in MATLAB Online, go to the Live Editor or Editor tab, and in the Navigate section, click Bookmark. Then, select Previous or Next.</p>	For information about setting and clearing bookmarks, see “Set Bookmarks” on page 24-27.

Note The Details panel does not display details for live scripts or live functions and is not available in MATLAB Online.

Set Bookmarks

You can set a bookmark at any line in a file in the Editor or Live Editor so that you can quickly navigate to the bookmarked line. This is particularly useful in long files. For example, suppose that 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.

To set a bookmark in the Editor, position the cursor on the line that you want to add the bookmark to.

Then, go to the **Editor** tab, and in the **Navigate** section, click  **Go To**. Under **Bookmarks**, select **Set/Clear**. A bookmark icon  appears to the left of the line. To clear the bookmark, with the cursor anywhere on the line with the bookmark, click  **Go To** and under **Bookmarks**, select **Set/Clear**.

To set a bookmark in the Live Editor and in MATLAB Online, with the cursor on the line that you want to add the bookmark to, go to the **Live Editor** or **Editor** tab, and in the **Navigate** section, click  **Bookmark**. To clear the bookmark, click **Bookmark**, and select **Set/Clear**. You also can click the bookmark icon  to the left of the line.

MATLAB does not maintain bookmarks after you close a file.

Navigate Backward and Forward in Files

In the Editor, you can access lines in a file in the same sequence that you previously navigated or edited them. To navigate backward and forward in sequence, on the **Editor** tab, in the **Navigate** section, click the  and  buttons. Backward and forward navigation is not supported in the Live Editor or in MATLAB Online.

Clicking the  and  buttons interrupts the backward and forward sequence. Editing a line or navigating to another line using the list of features described in “Navigate to a Specific Location” on page 24-25 also interrupts the sequence.

Once the sequence is interrupted, 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 that you edit or navigate to after interrupting the sequence are added to the sequence after the interruption point.

For example, in the Editor, open a file containing more than 6 lines and edit lines 2, 4, and 6. Click the  button to return to line 4, and then again to return to line 2. Click the  button to return to line 4. Edit line 3. This interrupts the sequence. You can no longer use the  button to return to line 6. You can, however, click the  button to return to line 2.

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 or Live 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 or Live 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 or Live Editor runs the <code>open</code> 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.
Model	Opens in Simulink.
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.

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.

The screenshot shows a MATLAB window titled "TODO/FIXME Report". It displays a list of annotated lines from a file named "ACREALM.m". The annotations include TODO, FIXME, and NOTE markers. The report interface includes buttons for "Rerun This Report" and "Run Report on Current Folder", and a checkbox for selecting annotation types (TODO, FIXME, NOTE).

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 2-51. 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 case 1
11 output = pi * radius^2;
12 case 2
13 output = 4 * pi * radius^2;
14 % Fixme: Is the area of hemisphere as below?
15 % case 3
16 % output = 2 * pi * radius^2;
17 % FIXME
18 otherwise
19 disp('Incorrect flag')
20 output = NaN;
21 % NOTE: Find out from the manager if we need to include
22 % the area of a cone
23 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 24-31](#)

["Changing Code Based on Code Analyzer Messages" on page 24-32](#)

["Other Ways to Access Code Analyzer Messages" on page 24-32](#)

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:

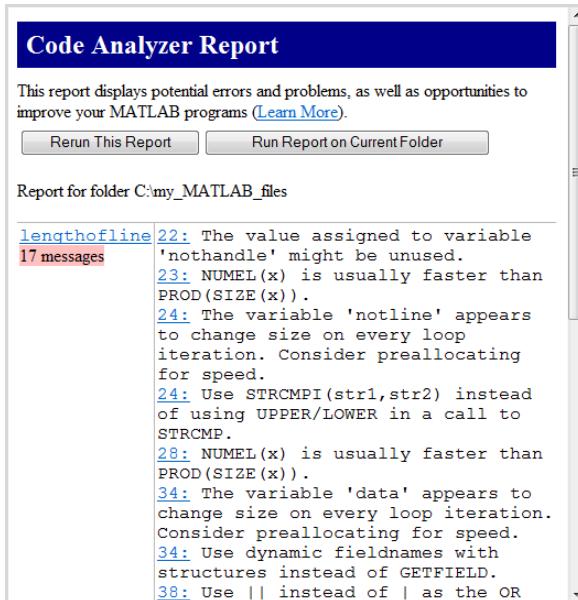
- 1 In the Current Folder browser, navigate to the folder that contains the files you want to check.

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`.

- 2 `copyfile(fullfile(matlabroot,'help','techdoc','matlab_env','examples','lengthofline.m'))`

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 24-5.

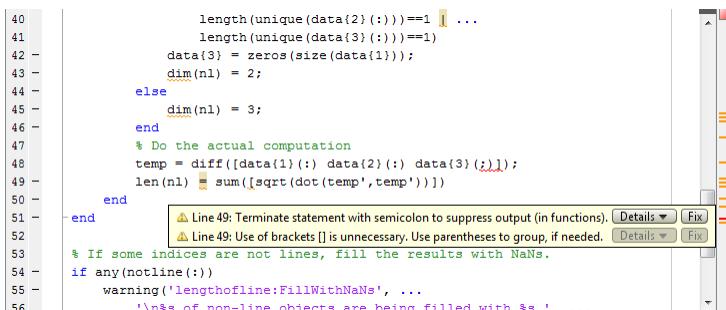
- If the message does not contain a link, and you are unsure what a message means or what to do, search for related topics in the Help browser. For examples of messages and what to do about them, including specific changes to make for the example, `lengthofline.m`, see “[Changing Code Based on Code Analyzer Messages](#)” on page 24-32.
 - 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 24-12.
 - 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 24-8.
- 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 Text Files](#)”.
- 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.



The screenshot shows a portion of the MATLAB Editor with the following code:

```

40 length(unique(data{2}(:)))==1 || ...
41 length(unique(data{3}(:)))==1
42 data{3} = zeros(size(data{1}));
43 dim(nl) = 2;
44 else
45 dim(nl) = 3;
46 end
47 % Do the actual computation
48 temp = diff([data{1}(:) data{2}(:) data{3}(1,:)]);
49 len(nl) = sum(sqrt(dot(temp',temp')));
50
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% of non-line objects are being filled with %s.' ...

```

Two tooltip windows are displayed at the bottom of the code area, both for line 49:

- The first tooltip says: “Line 49: Terminate statement with semicolon to suppress output (in functions).” It has a **Details** button and a **Fix** button.
- The second tooltip says: “Line 49: Use of brackets [] is unnecessary. Use parentheses to group, if needed.” It also has a **Details** button and a **Fix** button.

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 and Live Editor — Code Analyzer” on page 24-5.

MATLAB Code Compatibility Report

The Code Compatibility Report is a convenient tool that analyzes your code, lists the entire set of compatibility issues in tabular format, and provides you with instructions on how to address these compatibility issues. The report enables you to:

- Identify the compatibility issues that you must address for your code to run properly in the current MATLAB release.
- Estimate the effort required to update your code when you upgrade to a newer MATLAB release.
- Improve your code by replacing functionality that is not recommended.

The Code Compatibility Report displays the locations in your code that are affected by compatibility issues and provides links to the documentation for more information on how to make the necessary changes at each location.

Generate the Code Compatibility Report

To run the Code Compatibility Report:

- 1 In the Current Folder browser, navigate to and open the folder that contains the code files you want to analyze.
- 2 In the Current Folder browser that lists the files you want to analyze, either click  or right-click in the white space of the browser. Both options open a menu. Select **Reports > Code Compatibility Report**. Alternatively, you can run `codeCompatibilityReport` at the command prompt to generate the report.

The report displays in the MATLAB Web Browser, showing potential compatibility issues. For example:

The screenshot shows the MATLAB Code Compatibility Report interface. At the top, there are tabs for Overview, Syntax, Removal, Behavior, Not Supported, Upcoming Removals, and Upcoming Behavior. The Overview tab is selected. Below the tabs, there is a section for 'Analysis Date' (09-Jun-2020 09:40:41) and 'MATLAB Version' (R2020b). A note states that syntax errors are not incompatibilities, but result in nonrunnable code and impact compatibility analysis. It also mentions false positives and ongoing improvements. Under the 'Syntax Errors' section, there is a table showing two occurrences in 'exampleScript.m' and 'exampleFunction.m'. The 'Functionality that has been removed' section lists 'wavinfo' as removed, with a message to use 'AUDIOINFO' instead. The 'Unsupported functionality that might cause errors' section is empty.

- 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 most likely the code did not run properly in previous releases, syntax errors impact compatibility analysis. For example, *Parse error at '}': usage might be invalid MATLAB syntax*.
- 4 For each functionality listed in the report, review the issue description and your code. Messages include the line numbers to help locate the issue 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 **Documentation** link associated with the message.

Each functionality listed in the report displays a recommended action. You also can use the following general advice:

- **Functionality that has been removed** — Update your code to avoid compatibility errors in the current release.
- **Functionality that has changed behavior** — Confirm that the change in behavior is acceptable, and if not, update your code for the current release.
- **Unsupported functionality that might cause errors** — Files listed here use functionality that is unsupported, undocumented, and not intended for customer use. Update your code to use documented functionality to avoid errors and unexpected behavior changes.
- **Functionality that will be removed** — Update your code now or in a later release. Updating your code now makes future upgrades easier.
- **Functionality that will change behavior** — Investigate these changes now to make future upgrades easier.
- **New functionality that might improve code** — Consider updating your code. Current code is expected to continue working in future releases but newer functionality is recommended.

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.

Unsupported Functionality

The Code Compatibility Report checks for functionality that is unsupported, undocumented, and not intended for use. Such features are subject to change or removal without notice and can cause future errors. In some cases there is documented replacement functionality, but there might be no simple replacement. Contact MathWorks Support to describe your usage and request supported replacement.

See Also

`CodeCompatibilityAnalysis` | `analyzeCodeCompatibility` | `codeCompatibilityReport`

Programming Utilities

- “Identify Program Dependencies” on page 25-2
- “Protect Your Source Code” on page 25-6
- “Create Hyperlinks that Run Functions” on page 25-8
- “Create and Share Toolboxes” on page 25-11

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.

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 =  
  
'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.

The screenshot shows the 'Dependency Report' window. At the top, there are buttons for 'Rerun This Report' and 'Run Report on Current Folder'. Below these are three checkboxes: 'Show child functions' (checked), 'Show parent functions (current folder only)' (unchecked), and 'Show subfunctions' (unchecked). A note states 'Built-in functions and files in toolbox/matlab are not shown'. The main area is titled 'Report for Folder I:\my_MATLAB_files' and contains a table titled 'MATLAB File List'. The table has two columns: 'MATLAB File List' and 'Children (called functions)'. The data is as follows:

MATLAB File List	Children (called functions)
chirpy	toolbox : \images\images\erode.m toolbox : \shared\siglib\chirp.m toolbox : \signal\signal\specgram.m
collatz	
collatzall	subfunction : collatzplot new
collatzplot	current dir : collatz
collatzplot new	current dir : collatz
go	current dir : mobius
mobius	

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.

rational differential	toolbox	:	\rf\rf\s2sdd.m
	toolbox	:	\rf\rf\@rfckt\smith.m
	toolbox	:	\rf\rf\s2tf.m
	toolbox	:	\rf\rf\rationalfit.m
	toolbox	:	? Multiple class methods match freqresp.m
	toolbox	:	\rf\rf\@rfmodel\timeresp.m
	toolbox	:	\rf\rf\@rfckt\analyze.m
	toolbox	:	\signal\signal\fftfilt.m

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 25-6 — 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 25-7 — 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('<a href="matlab:a=3; b=4;c=hypot(a,b)">Hypotenuse</a>')
```

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 ``' 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('<a href="http://en.wikipedia.org/wiki/Hypotenuse">Hypotenuse</a>')
```

The result in the Command Window looks the same as the previous example, but instead opens a page at `en.wikipedia.org`:

Hypotenuse

Using `matlab:`, you can:

- “Run a Single Function” on page 25-8
- “Run Multiple Functions” on page 25-9
- “Provide Command Options” on page 25-9
- “Include Special Characters” on page 25-9

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('<a href="matlab:magic(4)">Generate magic square</a>')
```

It displays this link in the Command Window:

[Generate magic square](matlab:magic(4))

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('<a href="matlab: x=0:1:8;y=sin(x);plot(x,y)">Plot x,y</a>')
```

It displays this link in the Command Window:

[Plot x,y](matlab: x=0:1:8;y=sin(x);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('<a href = "matlab:state = 0">Disable feature</a>')
disp('<a href = "matlab:state = 1">Enable feature</a>')
```

The Command Window displays the links that follow. Depending on which link you click, MATLAB sets `state` to 0 or 1.

[Disable feature](matlab:state = 0)
[Enable feature](matlab:state = 1)

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('<a href="matlab:str = ''Value > 0'''>Positive</a>')
```

and generates the following hyperlink.

[Positive](matlab:str = ''Value > 0'''>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('<a href="matlab:str=[''Value '' char(62) '' 0'']">Positive</a>')
```

Create and Share Toolboxes

In this section...

["Create Toolbox" on page 25-11](#)

["Share Toolbox" on page 25-15](#)

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.
Author Name, Email, and Company	Enter contact information for the toolbox author. To save the contact information, click Set as default contact .
Toolbox Image	To select an image that represents your toolbox, click Select toolbox image .
Summary and Description	Enter the toolbox summary and description. It is good practice to keep the Summary text brief and to add detail to the Description 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.

Package a Toolbox Dialog Box Section	Description
Toolbox Files and Folders	<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 Exclude MATLAB script or function files with matching P-files option.</p> <p>To exclude other files or folders from the toolbox, register them in the text file that is displayed when you click Exclude files and folders. It is good practice to exclude any source control files related to your toolbox.</p>
Requirements	<p>Add-ons — List of add-ons required for your toolbox. Selected add-ons are downloaded and installed when the toolbox is installed. MATLAB auto-populates this list with the add-ons it thinks the toolbox requires and selects them all by default. You can choose to omit any add-ons you do not want to install with your toolbox.</p> <p>If MATLAB is unable to find the installation information for an add-on in the list, you must enter a download URL. The download URL is the location where MATLAB can download and install the add-on. When the toolbox is installed, MATLAB installs the add-on using the specified URL.</p> <p>External Files — List of the files required for your toolbox that are located outside the toolbox folder. MATLAB auto-populates this list with the files it thinks the toolbox requires and selects them all by default. You can choose to omit any files you do not want in your toolbox.</p>
Install Actions	<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 Manage the current MATLAB path. To reset the list to the default list, click Reset to the current MATLAB path.</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 Toolbox Dialog Box 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"> • Display Name — The name to display to the user when they install a toolbox. • License URL — 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. • Download URL — 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 Add Platform to add a download URL for additional platforms. <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 “Get and Manage 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> <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 <code>toolboxname\getInstallationLocation.mlx</code> link at the bottom of the Installation of Additional Software section and enter the installed location of each additional piece of software on your computer.</p>
Toolbox Portability	MATLAB uses the information in the Toolbox Portability 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.

Package a Toolbox Dialog Box Section	Description
	<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>
Examples, Apps, and Documentation	<p>Examples—Published MATLAB examples associated with your toolbox. To include .m and .mlx files as examples, click the Add examples button, select your code file, and click Publish HTML. 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"> For a live script (.mlx) example, export it to HTML. On the Live Editor tab, select Save > Export to HTML and save it in a folder named <code>html</code>. For a script (.m) example, publish it to HTML with the publish 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>).
	<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 Supplemental Software link at the bottom of the Help browser home page. For more information, see “Display Custom Examples” on page 31-29.</p>
	<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"> To specify which apps (.mlapp files) are also installed and registered in the user's MATLAB Apps Gallery, select the apps. All .mlappinstall files in your toolbox folder are installed and registered in the user's MATLAB Apps Gallery.

Package a Toolbox Dialog Box Section	Description
	<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 <code>GettingStarted.mlx</code> in a <code>doc</code> subfolder within your toolbox folder.</p> <p>Alternatively, you can generate and edit <code>GettingStarted.mlx</code> 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 “Get and Manage Add-Ons”.</p> <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 Supplemental Software link at the bottom of the Help browser home page. For more information, see “Display Custom Documentation” on page 31-21.</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 and Manage 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 and Manage 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 and Manage Add-Ons”
- “Display Custom Examples” on page 31-29
- “Package Apps From the MATLAB Toolstrip”
- “Display Custom Documentation” on page 31-21

Function Argument Validation

Function Argument Validation

Introduction to Argument Validation

Function argument validation is a way to declare specific restrictions on function input arguments. Using argument validation you can constrain the class, size, and other aspects of function input values without writing code in the body of the function to perform these tests.

Function argument validation is declarative, which enables MATLAB desktop tools to extract information about a function by inspection of specific code blocks. By declaring requirements for input arguments, you can eliminate cumbersome argument-checking code and improve the readability, robustness, and maintainability of your code.

The function argument validation syntax simplifies the process of defining optional, repeating, and name-value arguments. The syntax also enables you to define default values in a consistent way.

Where to Use Argument Validation

The use of function argument validation is optional in function definitions. Argument validation is most useful in functions that can be called by any code and where validity of the inputs must be determined before executing the function code. Functions that are designed for use by others can benefit from the appropriate level of restriction on argument inputs and the opportunity to return specific error messages based on the inputs to the functions.

Where Validation Is Not Needed

In local and private functions, and in private or protected methods, the caller is aware of input requirements, so these types of functions can be called with valid arguments.

Where Validation Is Not Allowed

You cannot use argument validation syntax in nested functions, abstract methods, or handle class destructor methods. For more information on argument validation in methods, see “Argument Validation in Class Methods” on page 26-15.

arguments Block Syntax

Functions define argument validation in code blocks that are delimited by the keywords `arguments` and `end`. If used, an `arguments` block must start before the first executable line of the function.

You can use multiple `arguments` blocks in a function, but all blocks must occur before any code that is not part of an `arguments` block.

The highlighted area in the following code shows the syntax for argument validation.

```

function myFunction(inputArg)
    arguments
        inputArg    (dim1,dim2,...)    ClassName    {fcn1,fcn2,...}    = defaultValue
    end
    % Function code
end

```

The diagram shows the syntax of a MATLAB function argument declaration. It highlights three optional parts: **Size** (in parentheses), **Class** (a single class name), and **Functions** (a comma-separated list of validation functions). The **Size** part is shown with a bracket below it. The **Class** and **Functions** parts are grouped together with a bracket below them.

The function argument declaration can include any of these kinds of restrictions:

- **Size** — The length of each dimension, enclosed in parentheses
- **Class** — The name of a single MATLAB class
- **Functions** — A comma-separated list of validation functions, enclosed in braces

You can also define a default value for the input argument in the function validation declaration for that argument. The default value must satisfy the declared restrictions for that argument.

Size

Validation size is the dimensions of the input argument, specified with nonnegative integer numbers or colons (:). A colon indicates that any length is allowed in that dimension. You cannot use expressions for dimensions. The value assigned to the argument in the function call must be compatible with the specified size, or MATLAB throws an error.

MATLAB indexed assignment rules apply to size specifications. For example, a 1-by-1 value is compatible with the size specified as (5,3) because MATLAB applies scalar expansion. Also, MATLAB row-column conversion applies so that a size specified as (1,:) can accept a size of 1-by-n and n-by-1.

Here are some examples:

- (1,1) — The input must be exactly 1-by-1.
- (3,:) — The first dimension must be 3, and second dimension can be any value.

If you do not specify a size, then any size is allowed unless restricted by validation functions.

Class

Validation class is the name of a single class. The value assigned to the function input must be of the specified class or convertible to the specified class. Use any MATLAB class or externally defined class that is supported by MATLAB, except Java, COM classes, and MATLAB class definitions that do not use the `classdef` keyword (classes defined before MATLAB software Version 7.6).

Here are some examples:

- `char` — Input must be of class `char`, or a value that MATLAB can convert to a `char`, such as `string`.

- **double** — Input can be a numeric value of any precision.
- **cell** — Input must be a cell array.
- A user-defined class, such as an enumeration class, can restrict inputs to more specific values and enables you to control what conversions are supported.

If you do not specify a class, then any class is allowed unless restricted by validation functions.

Validation Functions

A validation function is a MATLAB function that throws an error if certain requirements are not satisfied by the argument value. Validation functions do not return values and, unlike class and size, cannot change the value of the arguments they are validating.

During the validation process, MATLAB passes the argument value to each validation function listed for that argument. The value passed to the validation functions is the result of any conversion made by the class and size specifications. MATLAB calls each function from left to right and throws the first error encountered.

For a table of predefined validation functions, see “Argument Validation Functions” on page 26-21.

Default Value

An argument default value can be any constant or expression that satisfies the size, class, and validation function requirements. Specifying a default value in an argument declaration makes the argument optional. MATLAB uses the default value when the argument is not included in the function call. Default value expressions are evaluated each time the default is used.

Note Because MATLAB validates the default value only when the function is called without a value for the argument, an invalid default value causes an error only when the function is called without that argument.

Optional arguments must be positioned after required arguments in the function signature and in the `arguments` block. For more information on optional arguments, see “Required and Optional Positional Arguments” on page 26-6.

Validation Sequence

Arguments are validated from top to bottom in the `arguments` block. MATLAB validates each part of an argument declaration in a specific order. First the class is validated, then the size. The result of the class and size validations is passed to the validation functions. Each step is optional depending on whether class, size, and validation functions are in the argument declaration.

For more information, see “Order of Argument Validation” on page 26-15

Conversion to Declared Class and Size

Both class validation and size validation can change the value of an input argument because of standard MATLAB conversion rules. Therefore, the validated value in the function body can be different from the value passed when calling the function. The conversion rules are derived from the rules that MATLAB applies for indexed assignment of the form:

$A(indices) = value$

MATLAB can determine that the left side value has requirements for class and size and, in certain cases, can convert the right side value to be of the required class and size.

For related information, see “[Avoiding Class and Size Conversions](#)” on page 26-16.

Examples of Argument Validation

This `arguments` block specifies the size and class of the three inputs.

```
function out = myFunction(A, B, C)
    arguments
        A (1,1) string
        B (1,:) double
        C (2,2) cell
    end

    % Function code
    ...
end
```

In this function, the variables must meet these validation requirements:

- `A` is a string scalar.
- `B` is a 1-by-any length vector of doubles.
- `C` is a 2-by-2 cell array.

Value Conversion

The following function illustrates how inputs can be converted to match the classes specified in the `arguments` block. The `SpeedEnum` class is an enumeration class created to define the values allowed for the third input argument.

```
function forwardSpeed(a,b,c)
    arguments
        a double
        b char
        c SpeedEnum
    end

    % Function code
    disp(class(a))
    disp(class(b))
    disp(class(c))
end
```

Here is the enumeration class.

```
classdef SpeedEnum < int32
    enumeration
        Full    (100)
        Half    (50)
        Stop    (0)
    end
end
```

This call to the function uses input values that MATLAB can convert to the declared types. The actual argument types within the function are displayed as output.

```
forwardSpeed(int8(4), "A string", 'full')

double
char
SpeedEnum
```

Specific Restrictions Using Validation Functions

Validation functions can restrict input arguments in more specific ways. You can use predefined validation functions for many common kinds of validation, and you can define your own validation function to satisfy specific requirements.

For example, this function specifies the following validations using `mustBeNumeric`, `mustBeReal`, `mustBeMember`, and the local function `mustBeEqualSize`.

- Input `x` must be a real, numeric row vector of any length.
- Input `v` must be a real, numeric row vector the same size as `x`.
- Input `method` must be a character vector that is one of the three allowed choices. Because `method` specifies a default value, this argument is optional.

```
function myInterp(x,v,method)
    arguments
        x (1,:) {mustBeNumeric,mustBeReal}
        v (1,:) {mustBeNumeric,mustBeReal,mustBeEqualSize(v,x)}
        method (1,:) char {mustBeMember(method,['linear','cubic','spline'])} = 'linear'
    end
    % Function code
    ...
end

% Custom validation function
function mustBeEqualSize(a,b)
    % Test for equal size
    if ~isequal(size(a),size(b))
        eid = 'Size:notEqual';
        msg = 'Size of first input must equal size of second input.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Avoid using function argument validation within custom validation functions. For more information about defining validation functions and a list of predefined validation functions, see “Argument Validation Functions” on page 26-21.

Kinds of Arguments

Function argument validation can declare four kinds of arguments. Functions can define any of these kinds of arguments, but the arguments must be defined in the following order:

- 1 Required positional arguments
- 2 Optional positional arguments
- 3 Repeating positional arguments
- 4 Optional name-value arguments

Required and Optional Positional Arguments

Positional arguments must be passed to a function in a specific order. The position of the value passed in the argument list must correspond to the order that the argument is declared in the `arguments` block. All argument names in the `arguments` block must be unique.

Positional arguments in the `arguments` block are required when calling the function, unless the argument defines a default value. Specifying a default value in the argument declaration makes a positional argument optional because MATLAB can use the default value when no value is passed in the function call.

The default value can be a constant or an expression that produces a result that satisfies the argument declaration. The expression can refer to the arguments that are declared before it in the `arguments` block, but not arguments that are declared after it.

MATLAB evaluates the default value expression only when the argument is not included in the function call.

All optional arguments must be positioned after all required arguments in the `arguments` block. For example, in this argument block, `maxval` and `minval` have default values and are therefore optional.

```
function myFunction(x,y,maxval,minval)
    arguments
        x (1,:) double
        y (1,:) double
        maxval (1,1) double = max(max(x),max(y))
        minval (1,1) double = min(min(x),min(y))
    end

    % Function code
    ....
end
```

You can call this function with any of these syntaxes:

```
myFunction(x,y,maxval,minval)
myFunction(x,y,maxval)
myFunction(x,y)
```

An optional positional argument becomes required when its position must be filled in the function call to identify arguments that come after it. That is, if you want to specify a value for `minval`, you must specify a value for `maxval`.

Ignored Positional Arguments

MATLAB lets you ignore input arguments by passing a tilde character (~) in place of the argument. You can define a function that ignores unused positional arguments by adding a tilde character (~) in the `arguments` block corresponding to the position of the argument in the function signature. Add a tilde character (~) for each ignored argument in the function signature.

Ignored arguments cannot have default values or specify class, size, or validation functions.

The tilde character (~) is treated as an optional input argument unless it is followed by a required positional argument. For example, in this function the tilde character (~) represents an optional argument.

```
function c = f(~)
    arguments
        ~
    end

    % Function code
end
```

You can call this function with no arguments.

```
c = f
```

Or you can call this function with one argument.

```
c = f(2)
```

In the following function, the tilde character (~) represents a required argument.

```
function c = f(~,x)
    arguments
        ~
        x
    end

    % Function code
    ...
end
```

Calls to this function must include both arguments.

```
c = f(2,3)
```

For more information on calling functions with ignored inputs, see “Ignore Inputs in Function Definitions” on page 21-10.

Repeating Arguments

Repeating arguments are positional arguments that can be specified repeatedly as input arguments. Declare repeating arguments in an `arguments` block that includes the `Repeating` attribute.

```
arguments (Repeating)
    arg1
    arg2
    ...
end
```

Functions can have only one `Repeating arguments` block, which can contain one or more repeating arguments.

A function that defines a `Repeating arguments` block can be called with zero or more occurrences of all the arguments in the block. If a call to a function includes repeating arguments, then all arguments in the `Repeating arguments` block must be included for each repetition.

For example, if a `Repeating arguments` block defines arguments `x` and `y`, then each repetition must contain both `x` and `y`.

Repeating arguments cannot specify default values and therefore cannot be optional. However, you can call the function without including any repeating arguments.

Functions must declare repeating arguments after positional arguments and before name-value arguments. You cannot specify name-value arguments within a `Repeating` block. For information on name-value arguments, see “Name-Value Arguments” on page 26-10.

In the function, each repeating argument becomes a cell array with the number of elements equal to the number of repeats passed in the function call. The validation is applied to each element of the cell

array. If the function is called with zero occurrences of this argument, the cell array has a size of 1-by-0. That is, it is empty.

For example, this function declares a block of three repeating arguments, *x*, *y*, and *option*.

```
function [xCell,yCell,optionCell] = fRepeat(x,y,option)
    arguments (Repeating)
        x double
        y double
        option {mustBeMember(option,[ "linear", "cubic" ])}
    end

    % Function code
    % Return cell arrays
    xCell = x;
    yCell = y;
    optionCell = option;
end
```

You can call the function with no inputs or multiples of three inputs. MATLAB creates a cell array for each argument containing all the values passes for that argument. This call to *fRepeat* passes two sets of the three repeating arguments.

```
[xCell,yCell,optionCell] = fRepeat(1,2,"linear",3,4,"cubic")
```

```
xCell =
```

```
1×2 cell array
{[1]} {[3]}
```

```
yCell =
```

```
1×2 cell array
{[2]} {[4]}
```

```
optionCell =
```

```
1×2 cell array
{[ "linear" ]} {[ "cubic" ]}
```

The following function accepts repeating arguments for *x* and *y* inputs in a **Repeating arguments** block. In the body of the function, the values specified as repeating arguments are available in the cell arrays *x* and *y*. This example interleaves the values of *x* and *y* to match the required input to the *plot* function: *plot(x1,y1,...)*.

```
function myPlotRepeating(x,y)
    arguments (Repeating)
        x (1,:) double
        y (1,:) double
    end

    % Function code
    % Interleave x and y
```

```
z = reshape([x;y],1,[]);

% Call plot function
if ~isempty(z)
    plot(z{:});
end
end
```

Call this function with repeating pairs of input arguments.

```
x1 = 1:10;
y1 = sin(x1);
x2 = 0:5;
y2 = sin(x2);
myPlotRepeating(x1,y1,x2,y2)
```

Avoid Using varargin for Repeating Arguments

Using `varargin` with functions that use argument validation is not recommended. If `varargin` is restricted in size or class in the repeating arguments block, then the restrictions apply to all values in `varargin`.

If you use `varargin` to support legacy code, it must be the only argument in a `Repeating arguments` block.

For example, this function defines two required positional arguments and `varargin` as the repeating argument.

```
function f(a, b, varargin)
arguments
    a uint32
    b uint32
end
arguments (Repeating)
    varargin
end

% Function code
...
end
```

Name-Value Arguments

Name-value arguments associate a name with a value that is passed to the function. Name-value arguments:

- Can be passed to the function in any order
- Are always optional
- Must be declared after all positional and repeating arguments
- Cannot appear in an `arguments` block that uses the `Repeating` attribute
- Must use unique names even when using multiple name-value structures
- Cannot use names that are also used for positional arguments

Declare name-value arguments in an `arguments` block using dot notation to define the fields of a structure. For example, the structure named `NameValueArgs` defines two name-value arguments, `Name1` and `Name2`. You can use any valid MATLAB identifier as the structure name.

```
arguments
    NameValueArgs.Name1
    NameValueArgs.Name2
end
```

The structure name must appear in the function signature.

```
function myFunction(NameValueArgs)
```

Call the function using the field names in the name-value structure.

```
myFunction(Name1=value1,Name2=value2)
```

Before R2021a, pass names as strings or character vectors, and separate names and values with commas. Both syntaxes are valid in later releases.

The name of the structure used in the function signature is the name of the structure in the function workspace that contains the names and values passed to the function.

```
function result = myFunction(NameValueArgs)
    arguments
        NameValueArgs.Name1
        NameValueArgs.Name2
    end

    % Function code
    result = NameValueArgs.Name1 * NameValueArgs.Name2;
end

r = myFunction(Name1=3,Name2=7)

r =
```

21

Name-value arguments support partial name matching when there is no ambiguity. For example, a function that defines `LineWidth` and `LineStyle` as its two name-value arguments accepts `LineW` and `LineS`, but using `Line` results in an error. In general, using full names is the recommended practice to improve code readability and avoid unexpected behavior.

Default Values for Name-Value Arguments

You can specify a default value for each name. If you do not specify a default value and the function is called without that name-value argument, then that field is not present in the name-value structure. If no name-value arguments are passed to the function, MATLAB creates the structure, but it has no fields.

To determine what name-value arguments have been passed in the function call, use the `isfield` function.

For example, the following function defines two required positional arguments (`width` and `height`) and two name-value arguments (`LineStyle` and `LineWidth`). In this example, the `options` structure has two fields (`LineStyle` and `LineWidth`) containing either the default values or values specified as name-value arguments when the function is called.

```
function myRectangle(width,height,options)
    arguments
        width double
        height double
        options.LineStyle (1,1) string = "-"
        options.LineWidth (1,1) {mustBeNumeric} = 1
    end

    % Function code
    ...
end
```

All of these syntaxes are valid ways to call this function.

```
myRectangle(4,5)
myRectangle(4,5,LineStyle=":",LineWidth=2)
myRectangle(4,5,LineWidth=2,LineStyle=":")
myRectangle(4,5,LineStyle=":")
myRectangle(4,5,LineWidth=2)
```

Before R2021a, pass names as strings or character vectors, and separate names and values with commas. For example:

```
myRectangle(4,5,"LineStyle",":","LineWidth",2)
myRectangle(4,5,"LineWidth",2,"LineStyle",":")
```

Using Repeating and Name-Value Arguments

If the function defines repeating arguments, then you must declare the name-value arguments in a separate `arguments` block that follows the repeating arguments block. For example, this function accepts two repeating arguments, `x` and `y`. After specifying all repeats of `x` and `y`, you can specify a name-value argument that assigns the value `lin` or `log` to the `PlotType` name.

To determine if the function call includes the `PlotType` argument, use the `isfield` function to check for the `PlotType` field in the `scale` structure.

```
function linLog(x,y,scale)
    arguments(Repeating)
        x (1,:) double
        y (1,:) double
    end
    arguments
        scale.PlotType (1,1) string
    end
    z = reshape([x;y],1,[]);
    if isfield(scale,"PlotType")
        if scale.PlotType == "lin"
            plot(z{:})
        elseif scale.PlotType == "log"
            loglog(z{:})
        end
    end
end
```

Call this function with or without the name-value argument.

```
myLinLog(1:5,1:5)
myLinLog(1:5,1:5,1:10,1:100:1000)
myLinLog(1:5,1:5,1:10,1:100:1000,PlotType="log")
```

Before R2021a, pass names as strings or character vectors, and separate names and values with commas. For example:

```
myLinLog(1:5,1:5,1:10,1:100:1000,"PlotType","log")
```

Multiple Name-Value Structures

Function argument blocks can contain multiple name-value structures. However, the field names must be unique among all structures. This function has two name-value structures: `lineOptions` and `fillOptions`. These structures cannot have the same field names.

The arguments in the `myRectangle` function are:

- `width` and `height` are required positional arguments of type `double`.
- `lineOptions.LineStyle` is a scalar string with a default value of `"-"`.
- `lineOptions.LineWidth` is a scalar numeric with a default value of 1.
- `fillOptions.Color` is a string.
- `fillOptions.Pattern` has no restrictions on its value.

```
function myRectangle(width,height,lineOptions,fillOptions)
    arguments
        width double
        height double
        lineOptions.LineStyle (1,1) string = "-"
        lineOptions.LineWidth (1,1) {mustBeNumeric} = 1
        fillOptions.Color string
        fillOptions.Pattern
    end

    % Function Code
    ...
end
```

Name-Value Arguments from Class Properties

A useful function syntax in MATLAB uses the public properties of a class for the names of name-value arguments. To specify name-value arguments for all settable properties defined by a class (that is, all properties with public `SetAccess`), use this syntax in an `arguments` block:

```
structName.?ClassName
```

A function can use the `"structName.?ClassName"` syntax only once. Therefore, a function can define only one name-value structure that gets its field names from a class, even if using different classes and structure names.

If the class places restrictions on values that you can assign to the property by using property validation, then the function applies the validation to the individual name-value arguments. For information on property validation, see “Validate Property Values”.

For example, this function has two required arguments, `x` and `y` and accepts any public property name and value for the `matlab.graphics.chart.primitive.Bar` class.

```
function myBar(x,y,propArgs)
    arguments
        x (:,:) double
        y (:,:) double
        propArgs.?matlab.graphics.chart.primitive.Bar
    end
    propertyCell = namedargs2cell(propArgs);
    bar(x,y,propertyCell{:})
end
```

Call the function with the required inputs and any settable property name-value pairs.

```
x = [1,2,3;4,5,6];
y = x.^2;
myBar(x,y)
myBar(x,y,FaceColor="magenta",BarLayout="grouped")
```

Before R2021a, pass names as strings or character vectors, and separate names and values with commas. For example:

```
myBar(x,y, "FaceColor", "magenta", "BarLayout", "grouped")
```

Override Specific Properties

You can override the class property validation by redefining the property name with a specific name-value argument in the arguments block.

```
structName.?ClassName
structName.PropertyName (dim1,dim2,...) ClassName {fcn1, fcn2, ...}
```

The specific name-value argument validation overrides the validation defined by class for the individually specified property name.

For example, the following function defines name-value arguments as the properties of the `matlab.graphics.chart.primitive.Bar` class. The function also overrides the property name `FaceColor` to allow only these specific values: `red` or `blue`.

The `matlab.graphics.chart.primitive.Bar` class has a default value for `FaceColor` that is not one of the restricted values (`red` or `blue`). Therefore, the overriding declaration must assign a default value that satisfies the restriction placed by the `mustBeMember` validation function. That is, the default value must be `red` or `blue`.

This function converts the name-value structure to a cell array containing interleaved names and values using the `namedargs2cell` function.

```
function myBar(x,y,propArgs)
    arguments
        x (:,:) double
        y (:,:) double
        propArgs.?matlab.graphics.chart.primitive.Bar
        propArgs.FaceColor {mustBeMember(propArgs.FaceColor,{'red','blue'})} = "blue"
    end
    propertyCell = namedargs2cell(propArgs);
    bar(x,y,propertyCell{:})
end
```

Call the function using the two required arguments, `x` and `y`. Optionally pass any name-value pairs supported by the `bar` function and a value for `FaceColor` that can be either `red` or `blue`. Other values for `FaceColor` are not allowed.

```
x = [1,2,3;4,5,6];
y = x.^2;
myBar(x,y)
myBar(x,y,FaceColor="red",BarLayout="grouped")
```

Argument Validation in Class Methods

Validating method input arguments is useful in public methods because calls to the method might not originate in class code. You can use function argument validation in concrete class methods, including concrete methods defined in abstract classes. However, abstract methods (methods not implemented by the class) cannot define `arguments` blocks. For information on class methods, see “Methods” and “Abstract Classes and Class Members”.

If a `classdef` file includes method prototypes for methods defined in separate files, include the `arguments` blocks in the separate files defining the method. For more information on defining methods in separate files, see “Methods in Separate Files”.

Subclass methods do not inherit function argument validation. In subclass methods that override superclass methods, you can add the same argument validation to the subclass method as is used in the superclass method.

Handle class destructor methods cannot use argument validation. A method named `delete` in a handle subclass that includes an `arguments` block is not treated as a destructor. (In other words, it is not called by MATLAB when the object is destroyed). For more information on class destructor methods, see “Handle Class Destructor”.

Order of Argument Validation

When a function is called, MATLAB validates the input arguments in the order they are declared in the `arguments` block, from top to bottom. Each argument is fully validated before the next argument is validated. Therefore, any reference to a previously declared argument uses values that have been validated. Functions throw an error as a result of the first validation failure.

Validated values can be different from the original values passed as inputs when the function is called. For example, this function declares the inputs as class `uint32` values. The third input declaration assigns a default value equal to the product of the first two inputs.

```
function c = f(a, b,c)
    arguments
        a uint32
        b uint32
        c uint32 = a .* b
    end

    % Function code
    ...
end
```

Calling the function with inputs that are a different numeric class (for example, `double`) results in a conversion to `uint32`.

```
c = f(1.8,1.5)
```

Because the optional input argument `c` is not specified in the function call, MATLAB evaluates the default value and assigns it to `c` after converting `a` and `b` to `uint32` values. In this case, the conversion results in a value of 2 for both inputs. Therefore, the product of `a` times `b` is four.

```
c =
```

```
uint32
```

```
4
```

If you specify a value for the third input, then the function assigns a value to `c` and does not evaluate the default value expression.

```
c = f(1.8,1.5,25)
```

```
c =
```

```
uint32
```

```
25
```

Avoiding Class and Size Conversions

During the validation process, MATLAB applies class and then size validation before calling any validation functions. If a default value is used for an optional input that is omitted in the function call, this value is assigned to the argument before applying any steps in the validation process.

When an argument value that is passed to a function does not match the class and size required by the validation, MATLAB converts the value to the declared class and size when conversion is possible. However, conversion might not be the desired behavior.

Here are some examples of conversions that MATLAB can perform.

To satisfy numeric class restriction:

- A `char` value can be converted to its Unicode numeric value.
- A `string` can be converted to a number or `NaN` if the string is not a representation of a single number.

To satisfy size restriction:

- Scalar expansion can change size from scalar to nonscalar.
- A column vector can be converted to a row vector.

To eliminate the standard conversions performed by MATLAB from input argument validation, use validation functions instead of class and size restrictions. Calls to validation functions do not return values and cannot change the value of the input argument.

For example, this function restricts the first input to a two-dimensional array of any size that is of class `double`. The second input must be a 5-by-3 array of any class.

```
function f(a, b)
arguments
    a (:,:) double
    b (5,3)
end

% Function code
...
end
```

Because of standard MATLAB type conversion and scalar expansion, you can call this function with the following inputs and not receive a validation error.

```
f('character vector',144)
```

By default, MATLAB converts the elements of the character vector to their equivalent numeric value and applies scalar expansion to create a 5-by-3 array from the scalar value 144.

Using specialized validation functions can provide more specific input argument validation. For example, this function defines specialized validation functions that it uses in place of the class and size specifications for the first and second arguments. These local functions enable you to avoid input value conversions.

- `mustBeOfClass` restricts the input to a specific class without allowing conversion or subclasses. For a related function, see `mustBeA`.
- `mustBeEqualSize` restricts two inputs to be of equal size without allowing scalar expansion. For a related function, see `mustBeScalarOrEmpty`.
- `mustBeDims` restricts the input to be of a specified dimension without allowing transposing or scalar expansion. For a related function, see `mustBeVector`.

```
function fCustomValidators(a,b)
    arguments
        a {mustBeOfClass(a,'double'), mustBeDims(a,2)}
        b {mustBeEqualSize(b,a)}
    end

    % Function code
    ...
end

% Custom validator functions
function mustBeOfClass(input,className)
    % Test for specific class name
    cname = class(input);
    if ~strcmp(cname,className)
        eid = 'Class:notCorrectClass';
        msg = ['Input must be of class ', cname];
        throwAsCaller(MException(eid,msg))
    end
end

function mustBeEqualSize(a,b)
    % Test for equal size
    if ~isequal(size(a),size(b))
        eid = 'Size:notEqual';
        msg = 'Inputs must have equal size.';
        throwAsCaller(MException(eid,msg))
    end
end

function mustBeDims(input,numDims)
    % Test for number of dimensions
    if ~isequal(length(size(input)),numDims)
        eid = 'Size:wrongDimensions';
        msg = ['Input must have dimensions: ', num2str(numDims)];
        throwAsCaller(MException(eid,msg))
```

```
    end
end
```

The `mustBeOfClass` function enforces strict class matching by requiring a match with the name of the class. A more general approach could use the `isa` function to also match subclasses of the specified class.

The `mustBeEqualSize` and `mustBeDims` functions enforce the strict declarations for the input arguments.

```
fCustomValidators('character vector',144)
```

```
Invalid input argument at position 1. Input must be of class double
```

In this call, the number of dimensions of the first input is wrong, so the validation function returns a custom error message.

```
fCustomValidators(ones(2,2,4),144)
```

```
Invalid input argument at position 1. Input must have 2 dimensions
```

The `mustBeEqualSize` validator function restricts the second input to a specific dimension, which is provided in the error message.

```
fCustomValidators(ones(2,2),144)
```

```
Invalid input argument at position 2. Input must be of size [5 3]
```

For related predefined validation functions, see `mustBeA`, `mustBeFloat`, and `mustBeVector`.

nargin in Argument Validation

The `nargin` function returns the number of function input arguments given in the call to the currently executing function. When using function argument validation, the value returned by `nargin` within a function is the number of positional arguments provided when the function is called.

Repeating arguments are positional arguments and therefore the number of repeating arguments passed to the function when called is included in the value returned by `nargin`.

The value that `nargin` returns does not include optional arguments that are not included in the function call. Also, `nargin` does not count any name-value arguments.

Use `nargin` to determine if optional positional arguments are passed to the function when called. For example, this function declares three positional arguments and a name-value argument. Here is how the function determines what arguments are passed when it is called.

- `nargin` determines if the optional positional argument `c` is passed to the function with a `switch` block.
- `isfield` determines if the name-value argument for `Format` is passed to the function.

```
function result = fNargin(a,b,c,namedargs)
    arguments
        a (1,1) double
        b (1,1) double
        c (1,1) double = 1
    namedargs.Format (1,:) char
```

```

end

% Function code
switch nargin
    case 2
        result = a + b;
    case 3
        result = a^c + b^c;
end
if isfield(namedargs,"Format")
    format(namedargs.Format);
end
end

```

In this function call, the value of nargin is 2:

```

result = fNargin(3,4)
result =
    7

```

In this function call, the value of nargin is 3:

```

result = fNargin(3,4,7.62)
result =
    4.3021e+04

```

In this function call, the value of nargin is 3:

```

result = fNargin(3,4,7.62,Format="bank")
result =
    43020.56

```

Restrictions on Variable and Function Access

arguments blocks exist in the function's workspace. Any packages, classes, or functions added to the scope of the function using the `import` command are added to the scope of the `arguments` block.

The only variables visible to validator functions and default value expressions are the input variables already declared. In this function, the default value of c is derived from a and b.

```

function c = f(a,b,c)
    arguments
        a uint32
        b uint32
        c uint32 = a * b
    end

    % Function code
    ...
end

```

However, you cannot refer to input variables not yet declared in an `arguments` block. For example, using this declaration for argument `a` in the previous function is not valid because `b` and `c` have not been declared yet.

```
arguments
    a uint32 = b * c
    b uint32
    c uint32
end
```

Argument validation expressions can reference only previously declared, and therefore validated, arguments. Validation functions and default values for name-value arguments cannot access other name-value arguments.

Limitations on Functions in `arguments` Block

Any references to previously declared arguments must be visible in the text of validation functions and default values. To ensure code transparency, do not use functions that interact with the function workspace. Specifically, do not use nested functions or any of the functions listed in the following table in the `arguments` block.

<code>assignin</code>	<code>builtin</code>	<code>clear</code>
<code>dbstack</code>	<code>eval</code>	<code>evalc</code>
<code>evalin</code>	<code>exist</code>	<code>feval</code>
<code>input</code>	<code>inputname</code>	<code>load</code>
<code>nargin</code>	<code>narginchk</code>	<code>nargoutchk</code>
<code>save</code>	<code>whos</code>	<code>who</code>

These restrictions apply only within the `arguments` block and do not apply to variables or functions in the body of the function.

Debugging Arguments Blocks

While debugging inside of an arguments block the workspace is *read only*. This means that it is possible to inspect the workspace and view the values assigned to variables. However, it is not possible to create new variables or change the values assigned to existing variables while the workspace is read only. Once the debugger is outside of the arguments block it will once again be possible to create or edit variables.

See Also

`arguments` | `namedargs2cell`

Related Examples

- “Argument Definitions”
- “Argument Validation Functions” on page 26-21
- “Validate Property Values”

Argument Validation Functions

MATLAB defines functions for use in argument validation. These functions support common use patterns for validation and provide descriptive error messages. The following tables categorize the MATLAB validation functions and describe their use.

Numeric Value Attributes

Name	Meaning	Functions Called on Inputs
mustBePositive(value)	value > 0	gt, isreal, isnumeric, islogical
mustBeNonpositive(value)	value ≤ 0	ge, isreal, isnumeric, islogical
mustBeNonnegative(value)	value ≥ 0	ge, isreal, isnumeric, islogical
mustBeNegative(value)	value < 0	lt, isreal, isnumeric, islogical
mustBeFinite(value)	value has no NaN and no Inf elements.	isfinite
mustBeNonNaN(value)	value has no NaN elements.	isnan
mustBeNonzero(value)	value $\sim= 0$	eq, isnumeric, islogical
mustBeNonsparse(value)	value has no sparse elements.	issparse
mustBeReal(value)	value has no imaginary part.	isreal
mustBeInteger(value)	value == floor(value)	isreal, isfinite, floor, isnumeric, islogical
mustBeNonmissing(value)	value cannot contain missing values.	ismissing

Comparison with Other Values

Name	Meaning	Functions Called on Inputs
mustBeGreater Than(value, c)	value > c	gt, isscalar, isreal, isnumeric, islogical
mustBeLessThan(value, c)	value < c	lt, isreal, isnumeric, islogical
mustBeGreaterThanOrEqual(value, c)	value >= c	ge, isreal, isnumeric, islogical
mustBeLessThanOrEqual(value, c)	value <= c	le, isreal, isnumeric, islogical

Data Types

Name	Meaning	Functions Called on Inputs
mustBeA(value, classnames)	value must be of specific class.	Uses class definition relationships
mustBeNumeric(value)	value must be numeric.	isnumeric
mustBeNumericOrLogical(value)	value must be numeric or logical.	isnumeric, islogical
mustBeFloat(value)	value must be floating-point array.	isfloat
mustBeUnderlyingType(value, typename)	value must have specified underlying type.	isUnderlyingType

Size

Name	Meaning	Functions Called on Inputs
mustBeNonempty(value)	value is not empty.	isempty
mustBeScalarOrEmpty(value)	value must be a scalar or be empty.	isscalar, isempty
mustBeVector(value)	value must be a vector.	isvector

Membership and Range

Name	Meaning	Functions Called on Inputs
mustBeMember(value,S)	value is an exact match for a member of S.	ismember
mustBeInRange(value,lower,upper,boundflags)	value must be within range.	gt, ge, lt, le

Text

Name	Meaning	Functions Called on Inputs
mustBeFile(path)	path must refer to a file.	isfile
mustBeFolder(folder)	path must refer to a folder.	isfolder
mustBeNonzeroLengthText(value)	value must be a piece of text with nonzero length.	Not applicable
mustBeText(value)	value must be a string array, character vector, or cell array of character vectors.	Not applicable
mustBeTextScalar(value)	value must be a single piece of text.	Not applicable
mustBeValidVariableName(varname)	varname must be a valid variable name.	isvarname

Define Validation Functions

Validation functions are MATLAB functions that check requirements on values entering functions or properties. Validation functions determine when to throw errors and what error messages to display.

Functions used for validation have these design elements:

- Validation functions do not return outputs or modify program state. The only purpose is to check the validity of the input value.
- Validation functions must accept the value being validated as an input argument. If the function accepts more than one input argument, the first input is the value to be validated.
- Validation functions rely only on the inputs. No other values are available to the function.
- Validation functions throw an error if the validation fails. Using `throwAsCaller` to throw exceptions avoids showing the validation function itself in the displayed error message.

Creating your own validation function is useful when you want to provide specific validation that is not available using the MATLAB validation functions. You can create a validation function as a local

function within the function file or place it on the MATLAB path. To avoid a confluence of error messages, do not use function argument validation within user-defined validation functions.

For example, the `mustBeRealUpperTriangular` function restricts the input to real-valued, upper triangular matrices. The validation function uses the `isTriu` and `isReal` functions.

```
function mustBeRealUpperTriangular(a)
    if ~(isTriu(a) && isreal(a))
        eidType = 'mustBeRealUpperTriangular:notRealUpperTriangular';
        msgType = 'Input must be a real-valued, upper triangular matrix.';
        throwAsCaller(MException(eidType,msgType))
    end
end
```

If the input argument is not of the correct type, the function throws an error.

```
a = [1 2 3+2i; 0 2 3; 0 0 1];
mustBeRealUpperTriangular(a)

Input must be a real-valued, upper triangular matrix.
```

See Also

More About

- “Function Argument Validation” on page 26-2

Ways to Parse Function Inputs

MATLAB is dynamically typed, which means that variables do not have a declared type and can hold different types of values. However, values always belong to a specific type, and a program can always query the class and size of a variable's current value.

Function input arguments are variables in the function workspace whose values come from the calling code or command-line users. When a function is widely used by others, it should determine if the input values match the values expected by the code in the function.

Argument checking allows the function to provide more useful information when input values are unexpected and the function is unable to perform as intended. MATLAB provides several ways to simplify the process of checking and processing function inputs.

Function Argument Validation

Many functions in MATLAB use one of these patterns for input arguments:

- One or more required input arguments
- One or more required input arguments followed by one or more optional input arguments
- One of the previous patterns followed by name-value pairs

An effective way to implement these common patterns is to declare the arguments using a `arguments` block, as described in “Function Argument Validation” on page 26-2. This syntax is new for release R2019b and does not work in earlier releases.

Function argument validation is a way to declare specific restrictions on function input arguments. It enables you to constrain the class, size, and other aspects of function input values without writing code in the body of the function to perform these tests.

validateattributes

The `validateattributes` function enables you to verify that the inputs to a function conform to a set of requirements. Call `validateattributes` for each input argument with parameters specifying argument requirements.

inputParser

For complex function signatures, use an `inputParser` object to express input argument requirements programmatically. An `inputParser` object parses and validates a set of inputs.

See Also

`arguments` | `inputParser` | `validateattributes`

More About

- “Function Argument Validation” on page 26-2
- “Check Function Inputs with `validateattributes`” on page 21-11
- “Parse Function Inputs” on page 21-13

Transparency in MATLAB Code

Code has transparent variable access if MATLAB can identify every variable access by scanning the code while ignoring comments, character vectors, and string literals. Variable access includes reading, adding, removing, or modifying workspace variables.

In these coding contexts, MATLAB requires transparent variable access:

- Function argument validation blocks. For more information, see “[Restrictions on Variable and Function Access](#)” on page 26-19
- The body of a `parfor` loop or `spmd` block. For more information, see “[Ensure Transparency in parfor-Loops or spmd Statements](#)” ([Parallel Computing Toolbox](#)).

In these contexts, nontransparent variable access results in run-time errors.

Writing Transparent Code

Transparent code refers to variable names explicitly. For example, in this code, MATLAB can identify `X` and `ii` as variables.

```
X = zeros(1,10);
for ii = 1:10
    X(ii) = randi(9,1);
end
```

However, in the following call to the `eval` function, MATLAB cannot recognize the variables in the statement that is passed to `eval` because the input is a character string.

```
X = zeros(1,10);
for ii = 1:10
    eval('X(ii) = randi(9,1);')
end
```

Before executing this code, MATLAB sees a call to the `eval` function with one argument, which is the character vector `'X(ii) = randi(9,1);'`.

To be transparent, code must refer to variable names explicitly so that MATLAB can identify the variables by inspection or static analysis. Using the `eval` function with the character vector `'X(ii) = randi(9,1);'` means that MATLAB must execute the code to identify `X` and `ii` as variables.

Here is a partial list of functions and coding that you cannot use with transparent variable access:

- `eval`, `evalc`, `evalin`, or `assignin`
- Scripts
- MEX functions that access the workspace variables dynamically, for example by using `mexGetVariable`
- Introspective functions such as `who` and `whos`
- The `save` and `load` commands, except when the result from `load` is assigned explicitly
- Any dynamic name reference

Passing a variable to a function using the command form is not transparent because it is equivalent to passing the argument as a character string. For example, these calls to the `clear` function are both nontransparent.

```
clear X  
clear('X')
```

If code creates workspace variables, but MATLAB can identify these new variables only after executing the code, then this code does not have transparent variable access. For example, MATLAB cannot determine what variables are loaded from a MAT file, so this statement is nontransparent.

```
load foo.mat
```

However, code that explicitly assigns the loaded variable to a name is transparent because MATLAB can recognize that the name on the left-hand side refers to a workspace variable. For example, this statement loads the variable X from the MAT file into the workspace in a variable named X.

```
X = load('foo.mat','X');
```

Access to variables must be transparent within the workspace. For example, code cannot use the `evalin` or `assignin` functions in a workspace that requires transparency to create variables in another workspace.

See Also

More About

- “Function Argument Validation” on page 26-2
- “Scope Variables and Generate Names”

Software Development

Error Handling

- “Exception Handling in a MATLAB Application” on page 27-2
- “Throw an Exception” on page 27-4
- “Respond to an Exception” on page 27-6
- “Clean Up When Functions Complete” on page 27-9
- “Issue Warnings and Errors” on page 27-13
- “Suppress Warnings” on page 27-16
- “Restore Warnings” on page 27-18
- “Change How Warnings Display” on page 27-20
- “Use try/catch to Handle Errors” on page 27-21

Exception Handling in a MATLAB Application

In this section...

- "Overview" on page 27-2
- "Getting an Exception at the Command Line" on page 27-2
- "Getting an Exception in Your Program Code" on page 27-3
- "Generating a New Exception" on page 27-3

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 49)  
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.

Use an `MException` object to access information about the exception in your program. For more information, see “Respond to an Exception” on page 27-6.

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.

Use an `MException` object to capture information about the error. For more information, see “Throw an Exception” on page 27-4.

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 an error 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` function for this.
- 4 If there is fix that can be suggested for the current error, you can add it to the `Correction` field of the `MException` that you intend to throw. Use the `addCorrection` function for this.
- 5 Use the `throw` or `throwAsCaller` function to have MATLAB 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.

Suggestions on How to Throw an Exception

This example illustrates throwing an exception using the steps just described.

Create a function, `indexIntoArray`, that indexes into a specified array using a specified index. The function catches any errors that MATLAB throws and creates an exception that provides general information about the error. When it catches an error, it detects whether the error involves the number of inputs or the specified index. If it does, the function adds additional exceptions with more detailed information about the source of the failure, and suggests corrections when possible.

```
function indexIntoArray(A,idx)

% 1) Detect the error.
try
    A(idx)
catch

    % 2) Construct an MException object to represent the error.
    errID = 'MYFUN:BadIndex';
    msg = 'Unable to index into array.';
    baseException = MException(errID,msg);

    % 3) Store any information contributing to the error.
    if nargin < 2
        causeException = MException('MATLAB:notEnoughInputs','Not enough input arguments.');
        baseException = addCause(baseException,causeException);
    end

    % 4) Suggest a correction, if possible.
    if(nargin > 1)
        exceptionCorrection = matlab.lang.correction.AppendArgumentsCorrection('1');
        baseException = baseException.addCorrection(exceptionCorrection);
    end

    throw(baseException);
end

try
    assert(isnumeric(idx),'MYFUN:notNumeric', ...
        'Indexing array is not numeric.')
catch causeException
    baseException = addCause(baseException,causeException);
end
```

```
if any(size(idx) > size(A))
    errID = 'MYFUN:incorrectSize';
    msg = 'Indexing array is too large.';
    causeException2 = MException(errID,msg);
    baseException = addCause(baseException,causeException2);
end

% 5) Throw the exception to stop execution and display an error
% message.
throw(baseException)
end
end
```

If you call the function without specifying an index, the function throws a detailed error and suggests a correction:

```
A = [13 42; 7 20];
indexIntoArray(A)

Error using indexIntoArray (line 21)
Unable to index into array.
```

Caused by:
Not enough input arguments.

Did you mean:
>> indexIntoArray(A, 1)

If you call the function with a nonnumeric index array that is too large, the function throws a detailed error.

```
A = [13 42; 7 20];
idx = ['a' 'b' 'c'];
indexIntoArray(A, idx)

Error using indexIntoArray (line 41)
Unable to index into array.
```

Caused by:
Error using indexIntoArray (line 25)
Indexing array is not numeric.
Indexing array is too large.

Respond to an Exception

In this section...

"Overview" on page 27-6

"The try/catch Statement" on page 27-6

"Suggestions on How to Handle an Exception" on page 27-7

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.

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: 49
```

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. `ME` is an `MException` object. When an exception occurs, MATLAB creates an `MException` object 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.

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.

Suggestions on How to Handle an Exception

The following example reads the contents of an image file. It includes detailed error handling, and demonstrates some suggested actions you can take in response to an error.

The image-reading function throws and catches errors in several ways.

- The first `if` statement checks whether the function is called with an input argument. If no input argument is specified, the program throws an error and suggests an input argument to correct the error.
- The `try` block attempts to open and read the file. If either the `open` or the `read` fails, the program catches the resulting exception and saves the `MException` object in the variable `ME1`.
- The `catch` block 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., `j peg` instead of `j pg`) 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)

% Check the number of input arguments.
if nargin < 1
    me = MException('MATLAB:notEnoughInputs','Not enough input arguments.');
    aac = matlab.lang.correction.AppendArgumentsCorrection('"image.png"');
    me = me.addCorrection(aac);
    throw(me);
end

% Attempt to read file and catch an exception if it arises.
try
    fid = fopen(filename,'r');
    d_in = fread(fid);
catch ME1
    % Get last segment of the error 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 to possible causes of the error. In this case, the function checks whether the identifier ends in '`InvalidFid`', indicating a file could not be found.
- Use a nested `try/catch` statement to retry the operation with improved input. In this case, the function retries the open and read operations using a known variation of the filename extension.
- Display an appropriate message.
- Add the first `MException` object to the `cause` field of the second.
- Add a suggested correction to an `MException` object.
- 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, close figures that remained open after the error occurred.

Clean Up When Functions Complete

In this section...

- “Overview” on page 27-9
- “Examples of Cleaning Up a Program Upon Exit” on page 27-10
- “Retrieving Information About the Cleanup Routine” on page 27-11
- “Using onCleanup Versus try/catch” on page 27-12
- “onCleanup in Scripts” on page 27-12

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
k = 3;
myFun
    function myFun
        fprintf('k is %d\n', k)
    end
end
```



```
function testCleanup
k = 3;
obj = onCleanup(@myFun);
function myFun
    fprintf('k is %d\n', k)
end
end
```

Examples of Cleaning Up a Program Upon Exit

Example 1 — Close Open Files on Exit

MATLAB closes the file with identifier `fid` when function `openFileSafely` terminates:

```
function openFileSafely(fileName)
fid = fopen(fileName, 'r');
c = onCleanup(@()fclose(fid));

s = fread(fid);
.

.

end
```

Example 2 — Maintain the Selected Folder

This example preserves the current folder whether `functionThatMayError` returns an error or not:

```
function changeFolderSafely(fileName)
    currentFolder = pwd;
    c = onCleanup(@()cd(currentFolder));

    functionThatMayError;
end % c executes cd(currentFolder) here.
```

Example 3 — Close Figure and Restore MATLAB Path

This example extends the MATLAB path to include files in the `toolbox\images` folders, and then displays a figure from one of these folders. After the figure displays, the cleanup routine `restore_env` closes the figure and restores the path to its original state.

```
function showImageOutsidePath(imageFile)
fig1 = figure;
imgpath = genpath([matlabroot '\toolbox\images']);
% Define the cleanup routine.
cleanupObj = onCleanup(@()restore_env(fig1, imgpath));
```

```
% Modify the path to gain access to the image file,
% and display the image.
addpath(imgpath);
rgb = imread(imageFile);
fprintf('\n    Opening the figure %s\n', imageFile);
image(rgb);
pause(2);

% This is the cleanup routine.
function restore_env(fighandle, newpath)
disp '    Closing the figure'
close(fighandle);
pause(2)

disp '    Restoring the path'
rmpath(newpath);
end
end
```

Run the function as shown here. You can verify that the path has been restored by comparing the length of the path before and after running the function:

```
origLen = length(path);

showImageOutsidePath('greens.jpg')
Opening the figure greens.jpg
Closing the figure
Restoring the path

currLen = length(path);
currLen == origLen
ans =
1
```

Retrieving Information About the Cleanup Routine

In Example 3 shown above, the cleanup routine and data needed to call it are contained in a handle to an anonymous function:

```
@()restore_env(fig1, imgpath)
```

The details of that handle are then contained within the object returned by the `onCleanup` function:

```
cleanupObj = onCleanup(@()restore_env(fig1, imgpath));
```

You can access these details using the `task` property of the cleanup object as shown here. (Modify the `showImageOutsidePath` function by adding the following code just before the comment line that says, "% This is the cleanup routine.")

```
disp '    Displaying information from the function handle:'
task = cleanupObj.task;
fun = functions(task)
wsp = fun.workspace{2,1}
fprintf('\n');
pause(2);
```

Run the modified function to see the output of the `functions` command and the contents of one of the `workspace` cells:

```
showImageOutsidePath('greens.jpg')

Opening the figure greens.jpg
Displaying information from the function handle:
fun =
    function: '@()restore_env(fig1,imgpath)'
        type: 'anonymous'
        file: 'c:\work\g6.m'
    workspace: {2x1 cell}
wsp =
    imageFile: 'greens.jpg'
    fig1: 1
    imgpath: [1x3957 char]
    cleanupObj: [1x1 onCleanup]
    rgb: [300x500x3 uint8]
    task: @()restore_env(fig1,imgpath)

Closing the figure
Restoring the path
```

Using onCleanup Versus try/catch

Another way to run a cleanup routine when a function terminates unexpectedly is to use a **try**, **catch** statement. There are limitations to using this technique however. If the user ends the program by typing **Ctrl+C**, MATLAB immediately exits the **try** block, and the cleanup routine never executes. The cleanup routine also does not run when you exit the function normally.

The following program cleans up if an error occurs, but not in response to **Ctrl+C**:

```
function cleanupByCatch
try
    pause(10);
catch
    disp('    Collecting information about the error')
    disp('    Executing cleanup tasks')
end
```

Unlike the **try/catch** statement, the **onCleanup** function responds not only to a normal exit from your program and any error that might be thrown, but also to **Ctrl+C**. This next example replaces the **try/catch** with **onCleanup**:

```
function cleanupByFunc
obj = onCleanup(@()...
    disp('    Executing cleanup tasks'));
pause(10);
```

onCleanup in Scripts

onCleanup does not work in scripts as it does in functions. In functions, the cleanup object is stored in the function workspace. When the function exits, this workspace is cleared thus executing the associated cleanup routine. In scripts, the cleanup object is stored in the base workspace (that is, the workspace used in interactive work done at the command prompt). Because exiting a script has no effect on the base workspace, the cleanup object is not cleared and the routine associated with that object does not execute. To use this type of cleanup mechanism in a script, you would have to explicitly clear the object from the command line or another script when the first script terminates.

Issue Warnings and Errors

In this section...

- “Issue Warnings” on page 27-13
- “Throw Errors” on page 27-13
- “Add Run-Time Parameters to Your Warnings and Errors” on page 27-14
- “Add Identifiers to Warnings and Errors” on page 27-14

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 ~strcmp(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 27-13. 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

An 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 27-16
- “Restore Warnings” on page 27-18
- “Exception Handling in a MATLAB Application” on page 27-2

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 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 27-18
- “Change How Warnings Display” on page 27-20

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 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 27-16
- “Clean Up When Functions Complete” on page 27-9

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</code> (terse)
<code>backtrace</code>	Display a stack trace after a warning is invoked.	<code>on</code> (enabled)

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: "folderthatisnotonpath" 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 27-2.

See Also

`MException` | `onCleanup` | `try`, `catch`

Program Scheduling

- “Schedule Command Execution Using Timer” on page 28-2
- “Timer Callback Functions” on page 28-4
- “Handling Timer Queuing Conflicts” on page 28-8

Schedule Command Execution Using Timer

In this section...

"Overview" on page 28-2

"Example: Displaying a Message" on page 28-2

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 28-4
 - “Handling Timer Queuing Conflicts” on page 28-8

Timer Callback Functions

In this section...

["Associating Commands with Timer Object Events" on page 28-4](#)

["Creating Callback Functions" on page 28-5](#)

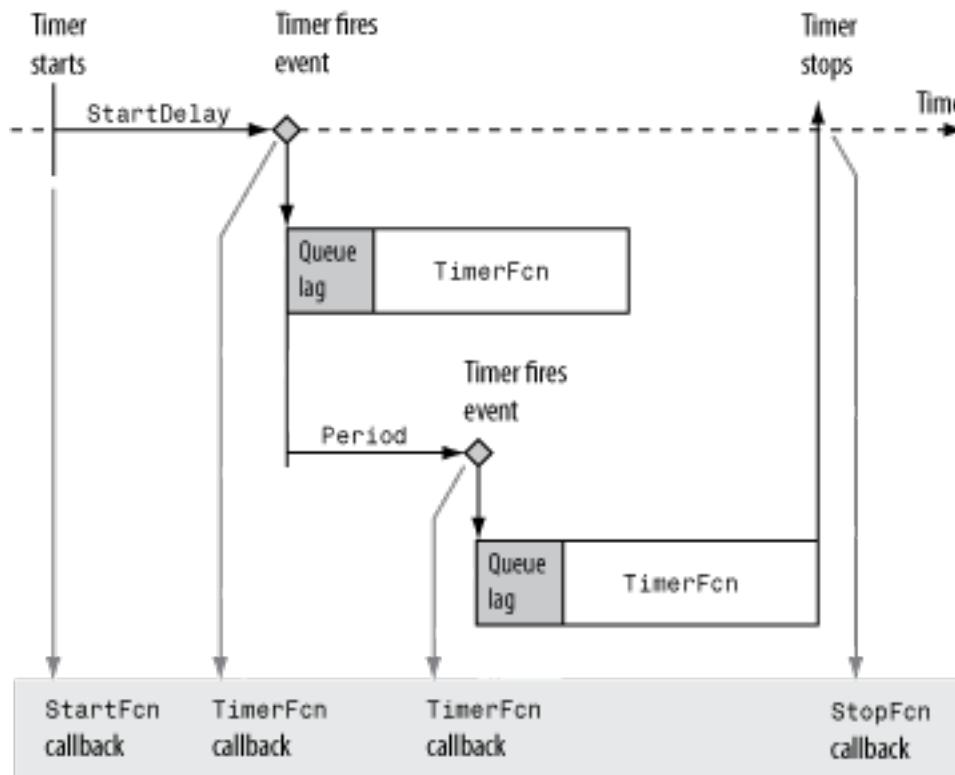
["Specifying the Value of Callback Function Properties" on page 28-6](#)

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 28-6.

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 28-6.

```
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 t
<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 28-5:

- 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 28-8

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 28-8
- “Error Mode” on page 28-9
- “Queue Mode” on page 28-10

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.

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, 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 TasksToExecute 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.
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 28-4

Performance

- “Measure the Performance of Your Code” on page 29-2
- “Profile Your Code to Improve Performance” on page 29-4
- “Determine Code Coverage Using the Profiler” on page 29-12
- “Techniques to Improve Performance” on page 29-14
- “Preallocation” on page 29-16
- “Vectorization” on page 29-18

Measure the Performance of Your Code

In this section...

- “Overview of Performance Timing Functions” on page 29-2
- “Time Functions” on page 29-2
- “Time Portions of Code” on page 29-2
- “The cputime Function vs. tic/toc and timeit” on page 29-2
- “Tips for Measuring Performance” on page 29-3

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 Your Code to Improve Performance” on page 29-4.

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 Your Code to Improve Performance” on page 29-4
- “Techniques to Improve Performance” on page 29-14
- MATLAB Performance Measurement White Paper on MATLAB Central File Exchange

Profile Your Code to Improve Performance

What Is Profiling?

Profiling is a way to measure the time it takes to run your code and identify where MATLAB spends the most time. After you identify which functions are consuming the most time, you can evaluate them for possible performance improvements. You also can profile your code to determine which lines of code do not run. Determining which lines of code do not run is useful when developing tests for your code, or as a debugging tool to help isolate a problem in your code.

You can profile your code interactively using the MATLAB Profiler or programmatically using the `profile` function. For more information about profiling your code programmatically, see `profile`. 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).

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.

Profile Your Code

To profile your code and improve its performance, use this general process:

- 1 Run the Profiler on your code.
- 2 Review the profile summary results.
- 3 Investigate functions and individual lines of code.

For example, you may want to investigate functions and lines of code that use a significant amount of time or that are called most frequently.

- 4 Save the profiling results.
- 5 Implement potential performance improvements in your code.

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.

- 6 Save the files, and run `clear all`. Run the Profiler again and compare the results to the original results.
- 7 Repeat the above steps to continue improving the performance of your code. 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.

Run the Profiler on Your Code

To run the Profiler on a line of code:

- 1 On the **Home** tab, in the **Code** section, click  **Run and Time** to open the Profiler. You also can type `profile viewer` in the Command Window.
- 2 Go to the **Profiler** tab, and in the **Profile** section, enter the code that you want to profile in the edit box.

For example, create a function `solvlotka.m` that finds the prey and predator population peaks for the Lotka-Volterra example provided with MATLAB:

```
function [preypeaks,predatorpeaks] = solvlotka(t0, tfinal, y0)
    [~,y] = ode23(@lotka,[t0 tfinal],y0);

    preypeaks = calculatepeaks(y(:,1));
    predatorpeaks = calculatepeaks(y(:,2));

end

function peaks = calculatepeaks(A)
    [TF,P] = islocalmax(A);
    peaks = P(TF);
end
```

Enter this statement in the edit box to profile the `solvlotka` function:

```
[preypeaks,predatorpeaks] = solvlotka(0,15,[20;20])
```

If you previously profiled the statement in the current MATLAB session, you also can select it from the edit box drop down list.

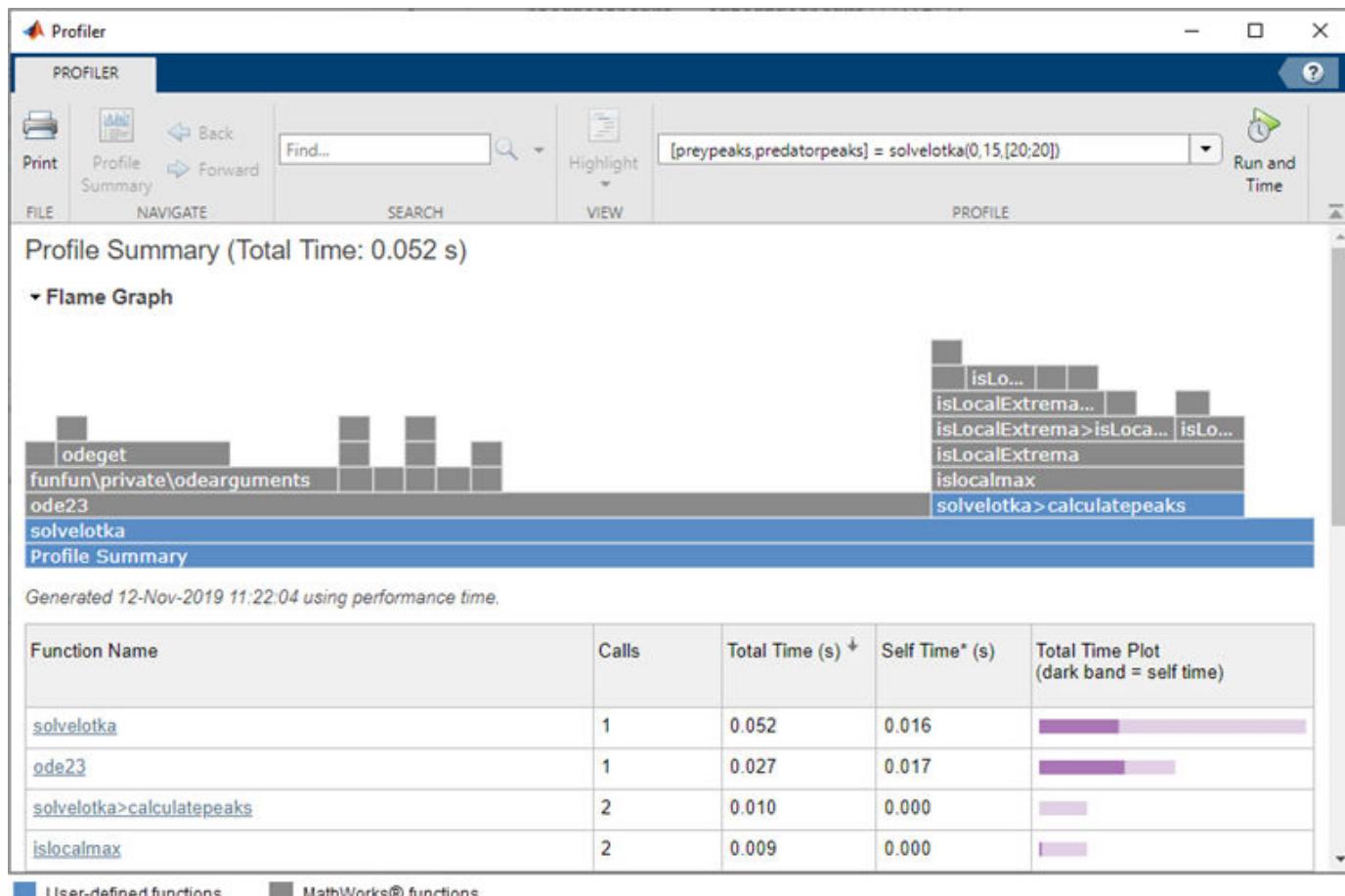
- 3**  Click **Run and Time**.

When profiling is complete, the Profiler displays the results in Profile Summary. The statements you profiled also display as having been run in the Command Window.

To profile a code file open in the Editor, on the **Editor** tab, in the **Run** section, click  **Run and Time**. The Profiler profiles the code file open in the current Editor tab and displays the results in the Profile Summary.

Review the Profile Summary Results

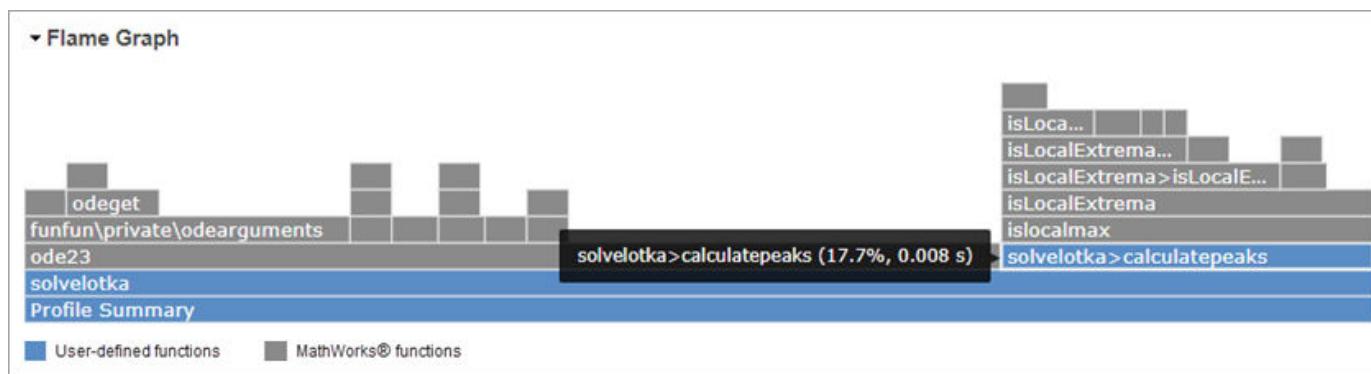
After running the Profiler on your code, the Profile Summary presents statistics about the overall execution of your code and provides summary statistics for each function called. For example, the image below shows the Profile Summary for the `solvlotka` function.



At the top of the Profile Summary results, a flame graph shows a visual representation of the time MATLAB spent running the code. Each function that was run is represented by a bar in the flame graph. User-defined functions display in blue, and MathWorks functions display in gray.

The functions in the graph display in hierarchical order, with parent functions appearing lower on the graph, and child functions appearing higher on the graph. The bar that spans the entire bottom of the graph labeled **Profile Summary** represents all of the code that ran. The width of a bar on the graph represents the amount of time it took for the function to run as a percentage of the total run time.

To see the actual percentage and time values as well as the full function name, hover over the bar in the graph. To display detailed information about a function including information about individual code lines, click the bar representing that function.



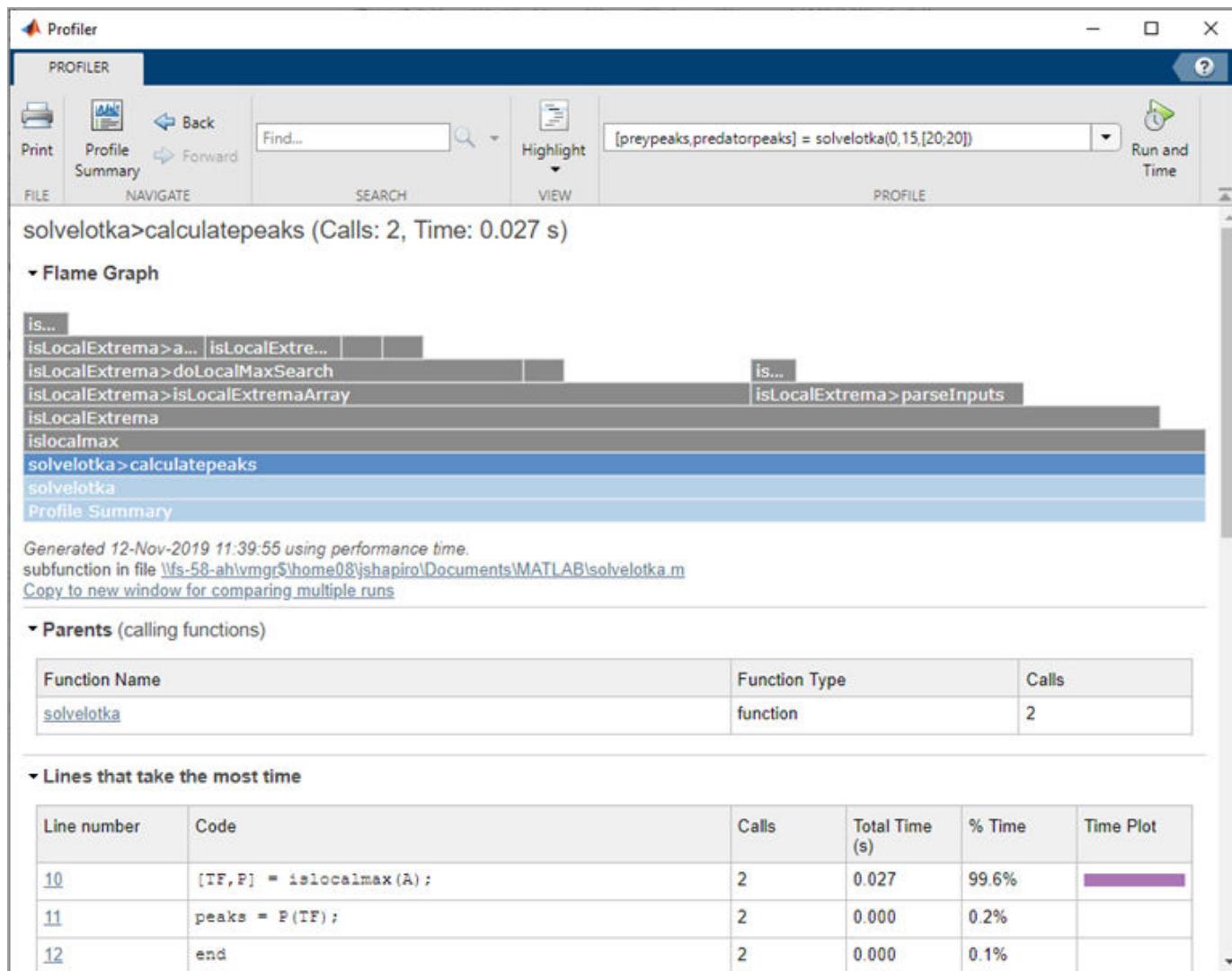
The function table below the flame frame displays similar information to the flame graph. Initially the functions appear in order of time they took to process. This table describes the information in each column.

Column	Description
Function Name	Name of the function called by the profiled code.
Calls	Number of times the profiled code called the function.
Total Time	Total time spent in the function, in seconds. The time for the 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.

To sort the function table by a specific column, click the arrow in the column header. For example, click the arrow in the **Function Name** column to sort the functions alphabetically. Initially the results appear in order by **Total Time**. To display detailed information about a function including information about individual code lines, click the function name.

Investigate Functions and Individual Code Lines

To find potential improvements in your code, look for functions in the flame graph or function table that use a significant amount of time or that are called most frequently. Click a function name to display detailed information about the function, including information about individual code lines. For example, click the `solveotka>calculatepeaks` function. The Profiler displays detailed information for the function.



At the top of the page, next to the name of the current function, the Profiler displays the number of times the function was called by a parent function and the total time spent in the function. Use the displayed links underneath the flame graph to open the function in your default editor or copy the displayed results to a separate window.

To return to the Profile Summary, in the **Profiler** tab, click the **Profile Summary** button. You also can click the **Profile Summary** bar at the bottom of the flame graph.

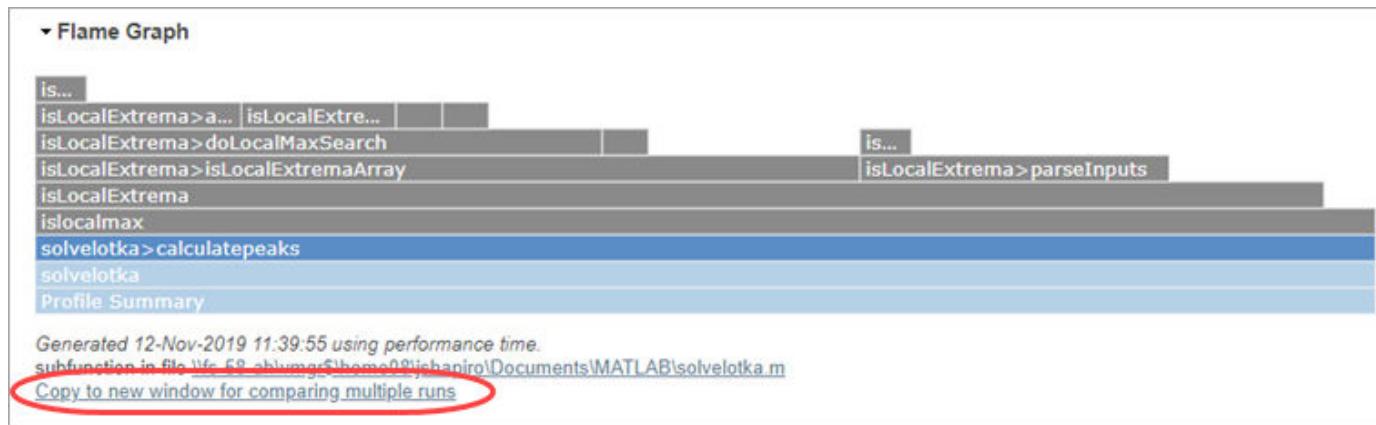
Once you have clicked an individual function, the Profiler displays additional information in these sections:

Section	Details
Flame Graph	<p>Flame graph showing visual representation of the time MATLAB spent running the profiled function. The graph shows the hierarchy of the profiled function, including child functions (displayed above the current function) and parent functions (displayed below the current function). User-defined functions display in blue (■), and MathWorks functions display in gray (■).</p> <p>Hover over the bar in the graph to see the actual percentage and time values as well as the full function name. Click a bar representing a function to display detailed information about the function.</p>
Parents	<p>List of functions that call the profiled function, including the number of times the parent function called the profiled function.</p> <p>Click a function name in the list to display detailed information about the function.</p>
Lines that take the most time	<p>List of code lines in the profiled function that used the greatest amount of processing time.</p> <p>Click a code line to see it in the Function Listing section, within the context of the rest of the function code.</p>
Children	<p>List of all the functions called by the profiled function.</p> <p>Click a function name in the list to display detailed information about the function.</p>
Code Analyzer results	List of problems and potential improvements for the profiled function.
Coverage results	Code coverage statistics about the lines of code in the function that MATLAB executed while profiling.

Section	Details
Function listing	<p>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. <p>Click a line number in the Function listing to open the function in your default editor.</p> <ul style="list-style-type: none"> 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 Profiler 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, go to the Profiler tab, and in the View section, click Highlight. Select from the available highlighting options. For example, to highlight lines of code that did not run, select the Non coverage option.</p>

Save Your Results

To compare the impact of changes after you have made improvements to your code, save your profiling results. To save your results, use the displayed link underneath the flame graph to copy the displayed results to a separate window.



You also can print your results from the Profiler by going to the **Profiler** tab and clicking the **Print** button.

Profile Multiple Statements in Command Window

To profile more than one statement in the Command Window:

- 1** Go to the Command Window and type `profile on`.
- 2** Enter and run the statements that you want to profile.
- 3** After running all of the statements, type `profile off`.
- 4** Open the Profiler by typing `profile viewer`. You also can go to the **Home** tab, and in the **Code** section, click  **Run and Time**.
- 5** Review the Profile Summary results.

Profile an App

You can profile apps that you create in App Designer. You also can profile apps that ship with MathWorks products, such as the Filter Design and Analysis tool included with Signal Processing Toolbox.

To profile an app:

- 1** Open the Profiler by going to the **Home** tab, and in the **Code** section, click  **Run and Time**. You also can type `profile viewer` in the Command Window.
- 2** In the **Profile** section of the Profiler toolbar, click  **Start Profiling**. Make sure that there is no code in the edit box to the right of the button.
- 3** Start the app.
- 4** Use the app.
- 5** When you are finished, click  **Stop Profiling** in the Profiler toolbar.
- 6** Review the Profile Summary results.

Note To exclude the app startup process in the profile, reverse steps 2 and 3. In other words, start the app before you click  **Start Profiling**.

See Also

`profile` | `profsave`

More About

- “Measure the Performance of Your Code” on page 29-2
- “Techniques to Improve Performance” on page 29-14
- “Determine Code Coverage Using the Profiler” on page 29-12

Determine Code Coverage Using the Profiler

When you run the Profiler on a file, some code might not run, such as a block containing an `if` statement.

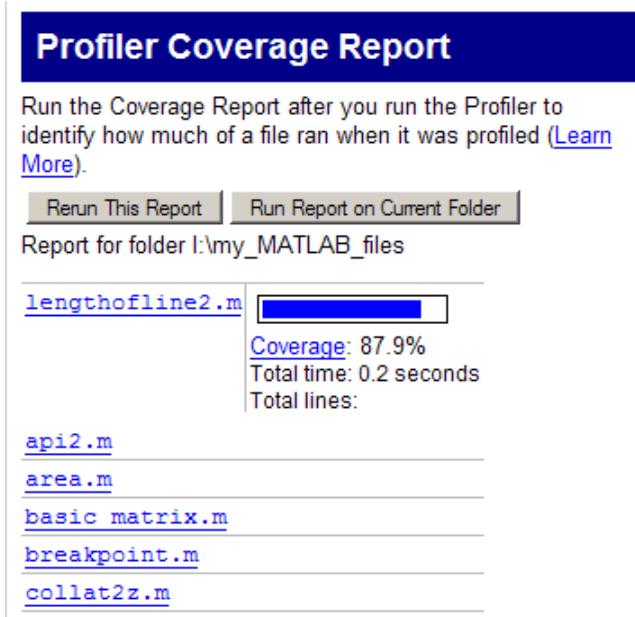
To determine how much of a file MATLAB executed when you profiled it, run the Coverage Report.

- 1 Profile your MATLAB code file. For more information, see “Profile Your Code to Improve Performance” on page 29-4 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`.



- 5 Click the **Coverage** link to see the Profile Detail Report for the file.

See Also

`profile`

More About

- “Profile Your Code to Improve Performance” on page 29-4
- “Measure the Performance of Your Code” on page 29-2
- “Techniques to Improve Performance” on page 29-14

Techniques to Improve Performance

In this section...

- “Environment” on page 29-14
- “Code Structure” on page 29-14
- “Programming Practices for Performance” on page 29-14
- “Tips on Specific MATLAB Functions” on page 29-15

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 29-16.
- Vectorize — Instead of writing loop-based code, consider using MATLAB matrix and vector operations. For more information, see “Vectorization” on page 29-18.
- 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, for example, in a MAT-file or .csv 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 the Performance of Your Code” on page 29-2
- “Profile Your Code to Improve Performance” on page 29-4
- “Preallocation” on page 29-16
- “Vectorization” on page 29-18
- “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
- `strings` for string arrays
- `cell` for cell arrays
- `table` for table 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 converts 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 12-15
- “Access Data Using Categorical Arrays” on page 8-24
- “Preallocate Arrays of Graphics Objects”
- “Construct Object Arrays”

More About

- “Techniques to Improve Performance” on page 29-14

Vectorization

In this section...

- “Using Vectorization” on page 29-18
- “Array Operations” on page 29-19
- “Logical Array Operations” on page 29-20
- “Matrix Operations” on page 29-21
- “Ordering, Setting, and Counting Operations” on page 29-22
- “Functions Commonly Used in Vectorization” on page 29-23

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 $A - \text{mean}(A)$, 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 2-25.

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 29-14
- “Array vs. Matrix Operations” on page 2-20

External Websites

- MathWorks Newsletter: Matrix Indexing in MATLAB

Memory Usage

- “Strategies for Efficient Use of Memory” on page 30-2
- “Resolve “Out of Memory” Errors” on page 30-6
- “How MATLAB Allocates Memory” on page 30-10
- “Avoid Unnecessary Copies of Data” on page 30-14

Strategies for Efficient Use of Memory

This topic explains several techniques to use memory efficiently in MATLAB.

Use 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 `mxArrays`) require room to store meta information about the data in memory, such as type, dimensions, and attributes. This takes about 104 bytes per array. This overhead only becomes an issue when you have a large number (e.g., hundreds or thousands) of small `mxArrays` (e.g., scalars). The `whos` command lists the memory used by variables, but does not include this overhead.

Because simple numeric arrays (comprising one `mxArray`) have the least overhead, you should use them wherever possible. When data is too complex to store in a simple array (or matrix), you can use other data structures.

Cell arrays are comprised of separate `mxArrays` for each element. As a result, cell arrays with many small elements have a large overhead.

Structures require a similar amount of overhead per field. 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. For more information, see “How MATLAB Allocates Memory” on page 30-10.

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 sparse and full storage requirements:

```
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 about 24 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)

Note that MATLAB supports most, but not all, mathematical operations on sparse arrays.

Avoid Temporary Copies of Data

You can significantly reduce the amount of memory required by avoiding the creation of unnecessary temporary copies of data.

Avoid Creating Temporary Arrays

Avoid creating large temporary variables, and also make it a practice to clear temporary variables when they are no longer needed. For example, this code creates an array of zeros stored as a temporary variable A, and then converts A to single-precision:

```
A = zeros(1e6,1);
As = single(A);
```

It is more memory efficient to use 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 non-double 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);
setrowval(400,0)
disp('The new value of A(399:401,1:10) is')
A(399:401,1:10)

    function setrowval(row,value)
        A(row,:) = value;
    end

end
```

Reclaim 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(1e5);  
b = rand(1e5);  
Out of memory.
```

More information

```
clear a  
a = rand(1e5); % New array
```

See Also

More About

- “Resolve “Out of Memory” Errors” on page 30-6

Resolve “Out of Memory” Errors

This topic explains several strategies you can use in situations where MATLAB runs out of memory. MATLAB is a 64-bit application that runs on 64-bit operating systems. It returns an error message whenever it requests a segment of memory from the operating system that is larger than what is available.

MATLAB has built-in protection against creating arrays that are too large. By default, MATLAB can use up to 100% of the RAM (not including virtual memory) of your computer to allocate memory for arrays, and if an array would exceed that threshold, then MATLAB returns an error. For example, this statement attempts to create an array with an unreasonable size:

```
A = rand(1e6,1e6);
```

```
Error using rand
Requested 1000000x1000000 (7450.6GB) array exceeds maximum array size preference. Creation of arrays greater than this limit may take a
MATLAB to become unresponsive.
```

[More information](#)

See “[Workspace and Variable Preferences](#)” for information on adjusting this array size limit. If you turn off the array size limit, then MATLAB returns a different error:

```
A = rand(1e6,1e6);
```

```
Out of memory.
```

[More information](#)

No matter how you run into memory limits, there are several resolutions available depending on your goals. The techniques discussed in “[Strategies for Efficient Use of Memory](#)” on page 30-2 can help you optimize the available memory you have, including:

- “[Use Appropriate Data Storage](#)” on page 30-2
- “[Avoid Creating Temporary Arrays](#)” on page 30-4
- “[Reclaim Used Memory](#)” on page 30-4

If you are already using memory efficiently and the problem persists, then the remaining sections of this page contain possible solutions.

Leverage tall Arrays

“[Tall Arrays for Out-of-Memory Data](#)” are designed to help you work with data sets that are too large to fit into memory. MATLAB works with small blocks of the data at a time, automatically handling all of the data chunking and processing in the background. There are two primary ways you can leverage tall arrays:

- 1 If you have a large array that fits in memory, but you run out of memory when you try to perform calculations, you can cast the array to a tall array:

```
B = tall(A)
```

This method enables you work with large arrays that can fit in memory, but that consume too much memory to allow for copies of the data during calculations. For example, if you have 8GB of RAM and a 5GB matrix, casting the matrix to a tall array enables you to perform calculations on the matrix without running out of memory. See “[Convert In-Memory Arrays to Tall Arrays](#)” for an example of this usage.

- 2 If you have file or folder-based data, you can create a `datastore` and then create a tall array on top of the datastore:

```
ds = datastore('path/to/data.csv');
tt = tall(ds);
```

This method gives you the full power of tall arrays in MATLAB: the data can have any number of rows and MATLAB does not run out of memory. And because `datastore` works with both local and remote data locations, the data you work with does not need to be on the computer you use to analyze it. See “Work with Remote Data” for more information.

Leverage the Memory of Multiple Machines

If you have a cluster of computers, you can use the Parallel Computing Toolbox and “Distributed Arrays” (Parallel Computing Toolbox) to perform calculations using the combined memory of all the machines in the cluster. This enables you to operate on the entire distributed array as a single entity. However, the workers operate only on their part of the array, and automatically transfer data between themselves when necessary.

Creating a distributed array is very similar to creating a tall array:

```
ds = datastore('path/to/data.csv');
dt = distributed(ds);
```

Load Only as Much Data as You Need

Another possible way to fix memory issues is to 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.

The `datastore` function enables you to work with large data sets incrementally. This function underpins “Tall Arrays for Out-of-Memory Data” and “Distributed Arrays” (Parallel Computing Toolbox), but you can use it for other purposes as well. Datastores are useful any time you want to load small portions of a data set into memory at a time.

To create a datastore, you need to provide the name of a file or a directory containing a collection of files with similar formatting. For example, with a single file:

```
ds = datastore('path/to/file.csv')
```

Or, with a collection of files in a folder:

```
ds = datastore('path/to/folder/')
```

You can also use the wildcard character * to select all files of a specific type, as in:

```
ds = datastore('data/*.csv')
```

Datastores support a wide variety of common file formats (tabular data, images, spreadsheets, and so on). See “Select Datastore for File Format or Application” for more information.

Aside from datastores, MATLAB also has several other functions to load parts of files, such as the `matfile` function to load portions of MAT-files. This table summarizes partial loading functions by file type.

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. See Big Data in MAT Files for an example of this usage.
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.
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>memmapfile</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".

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. Increasing the size of the swap file can increase the total available memory, but also typically leads to slower performance.

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. In the unlikely case you have set this preference, it must be large enough to accommodate:

- All the data to process
- MATLAB program files
- The MATLAB executable itself

- Additional state information

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 amount. 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.

See Also

`memory`

Related Examples

- “Strategies for Efficient Use of Memory” on page 30-2
- “Large Files and Big Data”
- “Work with Remote Data”
- “Java Heap Memory Preferences”

How MATLAB Allocates Memory

This topic provides information on how MATLAB® allocates memory when working with variables. This information, like any information on how MATLAB treats data internally, is subject to change in future releases.

Memory Allocation for Arrays

When you assign a numeric or character array to a variable, MATLAB allocates a contiguous 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 small, separate block of memory called a *header*. For most arrays, the memory required to store the header is insignificant. However, there could be some 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 fewer arrays require fewer array headers.

If you add new elements to an existing array, MATLAB expands the 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.

Copying Arrays

When you assign an array to a second variable (for instance, when you execute `B = A`), MATLAB does not allocate new memory right away. Instead, it creates a copy of the array reference. As long as you do not modify the contents of the memory block being referenced by `A` and `B`, there is no need to store more than one copy of data. However, if you modify any elements of the memory block using either `A` or `B`, MATLAB allocates new memory, copies the data into it, and then modifies the created copy.

On Windows® systems, the `memory` function enables you to inspect the memory details. To see how copying arrays affects memory usage on your Windows system, create the function `memUsed` in a file in your current folder. The function calls `memory` to return the amount of memory used by your MATLAB process in megabytes.

```
function y = memUsed
usr = memory;
y = usr.MemUsedMATLAB/1e6;
```

Call `memUsed` to display the current memory usage.

```
format shortG
memUsed

ans =
3966.1
```

Create a 2000-by-2000 numeric array and observe the change in memory usage. The array uses about 32 MB of memory.

```
A = magic(2000);
memUsed
```

```
ans =
    3998.1
```

Make a copy of A in B. Because 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 =
    3998.1
```

Now modify B by removing half of its rows. Because A and B no longer point to the same data, MATLAB must allocate a separate memory block to B. As a result, the amount of memory used by the MATLAB process increases by the size of B, which is about 16 MB (one half of the 32 MB required for A).

```
B(1001:2000,:) = [];
memUsed

ans =
    4014.1
```

Function Arguments

MATLAB handles arguments passed in function calls in the same way that it handles arrays being copied. When you pass a variable to a function, you actually pass a reference to the data that the variable represents. As long as the data is not modified by the called function, the variable in the calling function or script 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 variable in a new location in memory, updates that copy with the modified value, and points the input argument in the called function to this new location.

For example, consider the function `myfun`, which modifies the value of the array passed to it. MATLAB makes a copy of A in a new location in memory, sets the variable X as a reference to this copy, and then sets one row of X to zero. The array referenced by A remains unchanged.

```
A = magic(5);
myfun(A)

function myfun(X)
X(4,:) = 0;
disp(X)
end
```

If the calling function or script 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.

Data Types and Memory

Memory requirements differ for MATLAB data types. You might be able to reduce the amount of memory used by your code by learning how MATLAB treats various data types.

Numeric Arrays

MATLAB allocates 1, 2, 4, or 8 bytes to 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers, respectively. It represents floating-point numbers in either double-precision (`double`) or single-precision (`single`) format. Because MATLAB stores numbers of type `single` using 4 bytes, they require less memory than numbers of type `double`, which use 8 bytes. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`. In MATLAB, `double` is the default numeric data type and provides sufficient precision for most computational tasks. For more information, see “Floating-Point Numbers” on page 4-6.

Structure and Cell Arrays

While numeric arrays must be stored in a contiguous block of memory, structures and cell arrays can be stored in noncontiguous blocks. For structures and cell arrays, MATLAB creates a header not only for the array, but also for each field of the structure or each cell of the cell array. Therefore, 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, consider a scalar structure `S1` with fields `R`, `G`, and `B`, where each field contains a 100-by-50 array. `S1` requires one header to describe the overall structure, one header for each unique field name, and one header for each field. This makes a total of seven headers for the entire structure.

```
S1.R = zeros(100,50);
S1.G = zeros(100,50);
S1.B = zeros(100,50);
```

On the other hand, consider a 100-by-50 structure array `S2` in which each element has scalar fields `R`, `G`, and `B`. In this case, `S2` needs one header to describe the overall structure, one header for each unique field name, and one header for each field of the 5,000 elements, making a total of 15,004 array headers for the entire structure array.

```
for i = 1:100
    for j=1:50
        S2(i,j).R = 0;
        S2(i,j).G = 0;
        S2(i,j).B = 0;
    end
end
```

Use the `whos` function to compare the amount of memory allocated to `S1` and `S2` on a 64-bit system. Even though `S1` and `S2` hold the same data, `S1` uses significantly less memory.

```
whos S1 S2
```

Name	Size	Bytes	Class	Attributes
S1	1x1	120504	struct	
S2	100x50	1680192	struct	

Complex Arrays

MATLAB uses an interleaved storage representation of complex numbers, where the real and imaginary parts are stored together in a contiguous block of memory. If you make a copy of a complex array, and then modify only the real or imaginary part of the array, MATLAB creates an array containing both real and imaginary parts. For more information about the representation of complex numbers in memory, see “MATLAB Support for Interleaved Complex API in MEX Functions”.

Sparse Matrices

It is a good practice to store matrices with few nonzero elements using sparse storage. When a full matrix has a small number of nonzero elements, converting the matrix to sparse storage typically improves memory usage and code execution time. You can convert a full matrix to sparse storage using the `sparse` function.

For example, let matrix A be a 1,000-by-1,000 full storage identity matrix. Create B as a sparse copy of A. In sparse storage, the same data uses a significantly smaller amount of memory.

```
A = eye(1000);
B = sparse(A);
whos A B
```

Name	Size	Bytes	Class	Attributes
A	1000x1000	8000000	double	
B	1000x1000	24008	double	sparse

Working with Large Data Sets

When you work with large data sets, repeatedly resizing arrays might cause your program to run out of memory. If you expand an array beyond the available contiguous memory of its original location, MATLAB must make a copy of the array and move the copy into a memory block with sufficient space. During this process, 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. You can improve the memory usage and code execution time by preallocating the maximum amount of space required for the array. For more information, see “Preallocation” on page 29-16.

See Also

`memory` | `whos`

Related Examples

- “Strategies for Efficient Use of Memory” on page 30-2
- “Resolve “Out of Memory” Errors” on page 30-6
- “Avoid Unnecessary Copies of Data” on page 30-14

Avoid Unnecessary Copies of Data

In this section...

- “Passing Values to Functions” on page 30-14
- “Why Pass-by-Value Semantics” on page 30-17
- “Handle Objects” on page 30-17

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

When you do not need to preserve the original input values, you can assign the output of a function to the same variable that you provided as input.

```
A = f2(A);
```

In-place assignment follows the copy-on-write behavior described previously: modifying the input variable values results in a temporary copy of those values.

MATLAB can apply memory optimizations under certain conditions. Consider the following example. The `canBeOptimized` 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.

```
function canBeOptimized
A = rand(1e7,1);
A = fLocal(A);
end
function X = fLocal(X)
X = X.*1.1;
end
```

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. 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. Eliminating the copy in the local function saves memory and improves execution speed for large arrays.

However, MATLAB cannot apply this optimization if the assignment in the local function requires array indexing. For example, modifying the cell array created in `updateCells` requires indexing into the array `X` in the local function `gLocal`. The looped assignment in the form `X{i} = X{i}*1.1` creates a copy of the data in `X{i}`. For each element of the cell array, there are two copies in the workspace of `gLocal`.

```
function updateCells
C = num2cell(rand(1e7,1));
C = gLocal(C);
end
function X = gLocal(X)
for i = 1:length(X)
    X{i} = X{i}*1.1;
end
end
```

Several additional restrictions apply. 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 is directly accessible 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.

Debugging Code That Uses In-Place Assignment

When MATLAB applies in-place optimization to an assignment statement, the variable on the left side of the assignment is set to a temporary state that makes it inaccessible before MATLAB executes the right side of the assignment statement. If MATLAB stops in the debugger before the result of executing the right-side of the statement has been assigned to the variable, examining the left-side variable can produce an error indicating that the variable is unavailable.

For example, this function has a mismatch in the dimensions of variables `A` and `B`.

```
function A = inPlace
A = rand(100);
B = rand(99);
dbstop if error
A = A.*B;
end
```

Executing the function throws an error and stops in the debugger.

```
inPlace
Matrix dimensions must agree.

Error in inPlace (line 5)
A = A.*B;

5    A = A.*B;
```

Attempting to see the value of the variable `A` while in debug mode results in an error because the variable is temporarily unavailable.

```
K>> A
Variable "A" is inaccessible. When a variable appears on both sides of
an assignment statement, the variable may become temporarily
unavailable during processing.
```

To gain more flexibility when debugging, refactor your code to remove the in-place assignment. For example, assign the result to another variable.

```
function A = inPlace
A = rand(100);
B = rand(99);
dbstop if error
% Assign result to C instead of A
C = A.*B;
A = C;
end
```

Then the variable `A` is visible while in the debugger.

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 30-2

Custom Help and Documentation

- “Create Help for Classes” on page 31-2
- “Check Which Programs Have Help” on page 31-8
- “Create Help Summary Files — Contents.m” on page 31-10
- “Customize Code Suggestions and Completions” on page 31-12
- “Display Custom Documentation” on page 31-21
- “Display Custom Examples” on page 31-29

Create Help for Classes

In this section...

"Help Text from the doc Command" on page 31-2

"Custom Help Text" on page 31-3

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
```

MATLAB File Help: someClass [View code for someClass](#) [Default Topics](#)

someClass

`someClass` Summary of this class goes here
Detailed explanation goes here

Class Details

Sealed	false
Construct on load	false

Constructor Summary

[someClass](#) Summary of constructor

Property Summary

One	First public property
Two	Second public property

Method Summary

myMethod	Summary of myMethod
Static	myStaticMethod Summary of myStaticMethod

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 31-3
- “Methods” on page 31-4
- “Properties” on page 31-5
- “Enumerations” on page 31-5
- “Events” on page 31-6

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
```

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

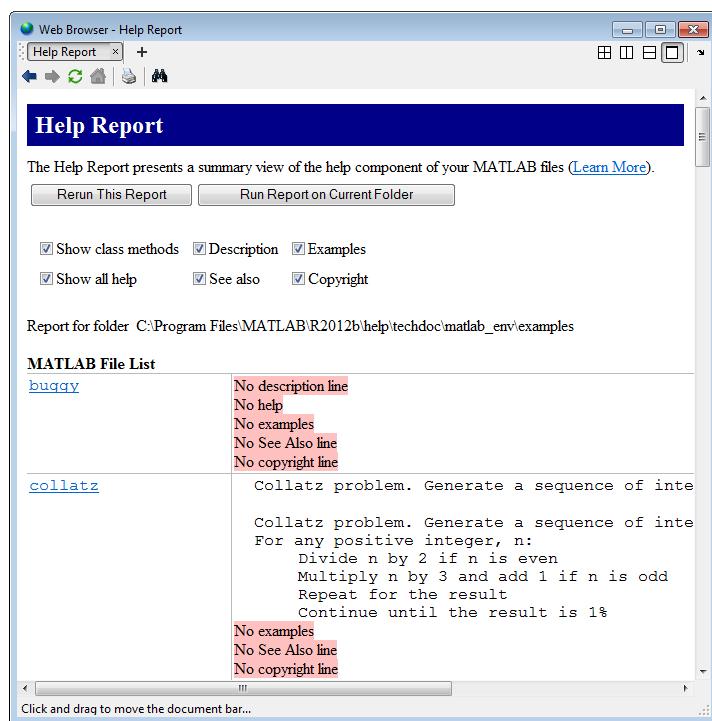
- “Role of Classes in MATLAB”
- “User-Defined Classes”

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.

Help Report Option	Description
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	<p>Check for an initial, nonempty comment line in the file. This line is sometimes called the H1 line.</p>
Examples	<p>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.</p>
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 20-5
- “Create Help Summary Files — `Contents.m`” on page 31-10

Create Help Summary Files – Contents.m

In this section...

- “What Is a Contents.m File?” on page 31-10
- “Create a Contents.m File” on page 31-10
- “Check an Existing Contents.m File” on page 31-11

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
mlintript    - 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 <code>Contents.m</code> 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 <code>Contents.m</code> with the first nonempty comment line in the corresponding file. Discrepancies appear in pink highlights. You can update either the program file or the <code>Contents.m</code> 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 describe 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. If you define information for a class method or package function, you must place `functionSignatures.json` in the parent folder of the outermost class or package folder. For example, to define information for a method of `myClass`, place `functionSignatures.json` in the folder `myFolder` for these class and package structures:

```
myFolder/+myPackage/@myClass
myFolder/+myPackage/+mySubpackage/@myClass
```

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":"nanflag", "kind":"flag", "type":["char", "choices {'includenan','omitnan'}"]}
    ]
  }

  "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 31-13.

```
{
  "functionName1": { signatureObj1 },
  "functionName2": { signatureObj2 }
}
```

To define information for a class method or package function, use the full name of the function or method. For example, define a class method `myMethod` and a package function `myFunction`.

```
{
  "myClass.myMethod": { signatureObj },
  "myPackage.myFunction": { signatureObj }
}
```

You can define multiple function signatures for the same function or method by defining multiple function objects with the same property (function or method name). For more information, see “Multiple Signatures” on page 31-19.

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 31-14.

```
{
  "functionName1":
  {
    "inputs": [ argumentObj1, argumentObj2 ]
  }
}
```

If you specify an instance method such as `myClass.myMethod` in the JSON file, one of the elements in `inputs` must be an object of `myClass`. Typically, this object is the first element. MATLAB supports code suggestions and completions for a specified method when you call it using either dot notation (`b = myObj.myMethod(a)`) or function notation (`b = myMethod(myObj,a)`) syntax.

Each signature in the JSON file can include the following properties.

Property	Description	JSON Data Type of Value
<code>inputs</code>	List of function input arguments. MATLAB uses this property for code suggestions and completions.	Array of argument objects
<code>outputs</code>	List of function output arguments. MATLAB uses this property to refine code suggestions and completions.	Array of argument objects

Property	Description	JSON Data Type of Value
platforms	<p>List of platforms that support the function. MATLAB does not present custom code suggestions and completions if the platform does not support the function.</p> <p>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".</p>	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": [[{"type": "numeric", "min": 0}, {"type": "containers.Map"}]]`".

Value	Argument Description
<code>"classname"</code>	Must be an object of class <code>classname</code> , where <code>classname</code> is the name of the class returned by the <code>class</code> function. For example, " <code>double</code> " or " <code>function_handle</code> ".
<code>"choices=expression"</code>	Must be a case-insensitive match to one of the specified choices. <code>expression</code> is any valid MATLAB expression that returns a cell of character vectors, string array, or cell of integer values. For example, " <code>choices={'on', 'off'}</code> " or " <code>choices={8, 16, 24}</code> ". <code>expression</code> can refer by name to other input arguments that appear in the argument list. Since <code>expression</code> is evaluated at run time, allowable choices can vary dynamically with the value of other input arguments.
<code>"file=*.ext,..."</code>	Must be a string or character vector that names an existing file with the specified extension. File names 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 " <code>file=*.m, *.mlx</code> ". To match all files in the current folder, use " <code>file</code> ".

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,size2,...,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=integerValue"	Must have a specified number of elements.
"nrows=integerValue"	Must have a specified number of rows.
"ncols=integerValue"	Must have a specified number of columns.
"numeric"	Must be numeric. A numeric value is one for which the <code>isa</code> function with the 'numeric' class category returns <code>true</code> .
"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 <code>double</code> .
"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.
>expression"	Must be numeric and satisfy the inequality. <i>expression</i> must return a full scalar double.
>=expression"	
<expression"	
<=expression"	
@args) expression	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 31-19.

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
```

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.

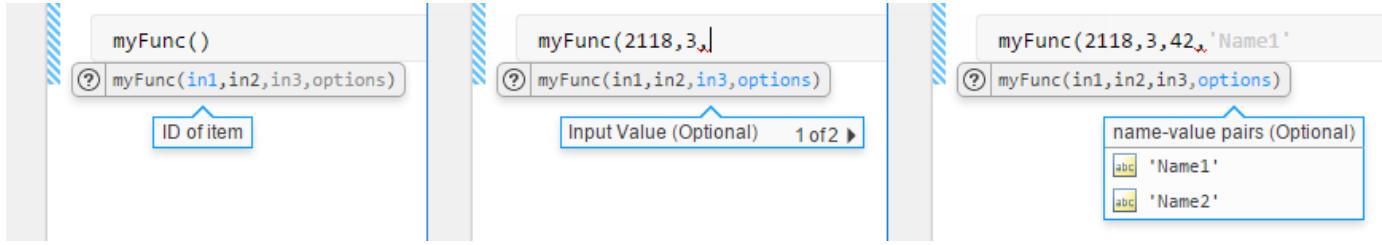
How Function Signature Information is Used

MATLAB uses the function signature information in the JSON file to display matching syntaxes as you type (*function hints*). You can complete partially typed text by pressing the **Tab** key (*tab completion*). In the Command Window or Editor, MATLAB does not use the JSON file to display function hints. The behavior of function hints and tab completion is summarized in the table.

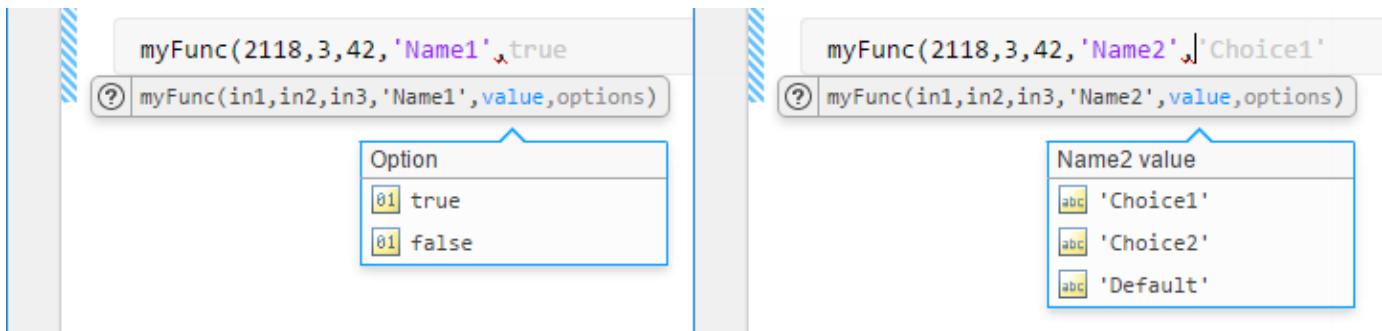
Tool	Function Hints	Tab Completion
Live Editor	Yes	Yes
Editor	No	Yes
Command Window	No	Yes

In MATLAB Online, the Editor behaves the same as the Live Editor.

To experiment with code suggestions in the Live Editor, start to call `myFunc` from a live script. 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 31-21
- "Create HTML Help Files" on page 31-22
- "Create info.xml File" on page 31-23
- "Create helptoc.xml File" on page 31-24
- "Build a Search Database" on page 31-26
- "Address Validation Errors for info.xml Files" on page 31-27

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 of the home page, under **Supplemental Software**, click the name of your toolbox. Your help opens in the current window.

The screenshot shows the MATLAB Documentation interface. At the top, there's a search bar labeled "Search Documentation" with a magnifying glass icon. Below the search bar, a navigation bar has tabs for "All", "Examples", "Functions", "Blocks", and "Apps". The "All" tab is selected. To the right of the tabs, there are links for "Edit Preferences" and "Release Notes" for "R2020a".

On the left, a sidebar titled "CONTENTS" lists categories under "Category": MATLAB, Simulink, Antenna Toolbox, Communications Toolbox, Control System Toolbox, Curve Fitting Toolbox, MATLAB Parallel Server, OPC Toolbox, Optimization Toolbox, and Parallel Computing Toolbox.

Under "Supplemental Software", it lists "Upslope Area Toolbox". This item is circled in red at the bottom of the sidebar.

The main content area features three large buttons: "MATLAB" (Explore MATLAB), "SIMULINK" (Explore Simulink), and a button with a wrench icon labeled "View Installation Help".

Below these buttons, there's a section titled "Applications" with five expandable categories: "Math, Statistics, and Optimization", "Signal Processing and Wireless Communications", "Control Systems", "Parallel Computing", and "Test and Measurement".

At the bottom of the main content area, there's a section titled "Hardware Support" with a note: "For a complete list of hardware solutions, see Hardware Support." Below this, there's another "Supplemental Software" section with the "Upslope Area Toolbox" listed again.

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 19-66.
- Create a script (*.m), and publish it to HTML. For more information, see “Publish and Share MATLAB Code” on page 23-2.

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">
  <xmstylesheet 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> .
<code><icon></code>	Icon for the Start button (not used)	none	No longer used, but the <code><icon></code> element is still required for MATLAB to parse the <code>info.xml</code> file.

XML Tag	Description	Value in Template	Notes
<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.

The screenshot shows the MATLAB Documentation interface. At the top, there's a blue header bar with the word "Documentation". Below it, a navigation bar has "CONTENTS" on the left and a "Close" button on the right. Underneath, a link "[All Products](#)" is visible. The main content area displays a search result for "My Toolbox (Supplemental Software)". This result is highlighted with a blue background. Below it, a section titled "Functions" is expanded, showing three items: "first", "second", and "third". At the bottom of this list is another link "[My Example](#)".

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.

To search for terms in your toolbox, open the Help browser, and in the **Search Documentation** field, enter the term you want to search for. Then, on the left side of the page, under **Refine by Source**, select **Supplemental Software** to view the results for your toolbox.

The screenshot shows the MATLAB Help browser search results for the term "tarboton". The search bar at the top contains "tarboton". On the left, there's a sidebar with a "FILTER" section. Under "Refine by Source", the "Supplemental Software" option is selected and circled in red. Other options include "MathWorks" (1 result) and "All Products". Below this is a "Refine by Product" section with "Upslope Area Toolbox" (9 results). The main content area shows search results. The first result is "dependenceMap", which is described as "A new method for the determination of flow directions and upslope areas in grid digital elevation models," from Water Resources Research, vol ... The Tarboton paper suggests that the dependence map can be calculated by repeated calculations of the Documentation > Upslope Area Toolbox. The second result is "upslopeArea", which is described as "A new method for the determination of flow directions and upslope areas in grid digital elevation models," from Water Resources Research, vol ... The Tarboton paper is not very specific about the handling of plateaus Documentation > Upslope Area Toolbox. Navigation arrows at the top right indicate results 1 to 9 of 9.

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 31-23.

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 31-29
- “Create and Share Toolboxes” on page 25-11
- “Add Help for Your Program” on page 20-5

Display Custom Examples

In this section...

"How to Display Examples" on page 31-29

"Elements of the demos.xml File" on page 31-30

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 19-66.
- Create a script (*.m), and publish it to HTML. For more information, see "Publish and Share MATLAB Code" on page 23-2.

- 2 Create a *demos.xml* file that describes the name, type, and display information for your examples. Place the file in the folder (or a subfolder of the folder) that contains your *info.xml* file. For more information about creating an *info.xml* file, see "Display Custom Documentation" on page 31-21.

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>
    </demosection>

</demos>
```

- 3 View your examples.

In the Help browser, navigate to the home page. At the bottom of the page, under **Supplemental Software** click the link for your example. Your example opens in the main help window.

If your examples do not display under **Supplemental Software**, the `demos.xml` file might contain an invalid construct.

Elements of the `demos.xml` File

Within the `demos.xml` file, you can include general information in the `<demos>` tag, define individual examples using the `<demoitem>` tag, and optionally define categories using the `<demosection>` tag.

Include General Information Using `<demos>` Tag

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.
<code><description></code>	The description that appears on the main page for your examples. Starting in R2021a, character data is not supported in the description of the <code>demos.xml</code> file. If your <code>demos.xml</code> file contains character data such as <code>&lt;</code> , <code>&gt;</code> , <code>&apos;</code> , <code>&quot;</code> , and <code>&amp;</code> in the description, the description does not appear correctly in the Help browser. To automatically replace existing character data with non-character data, use the <code>patchdemoxmlfile</code> function.
<code><website></code>	(Optional) Link to a website. For example, MathWorks examples include a link to the product page at https://www.mathworks.com .

Define Examples Using `<demoitem>` Tag

XML Tag	Notes
<code><label></code>	Defines the title to display in the browser.
<code><type></code>	Possible values are <code>M-file</code> , <code>model</code> , <code>M-GUI</code> , <code>video</code> , or <code>other</code> . Typically, if you published your example using the <code>publish</code> function, the appropriate <code><type></code> is <code>M-file</code> .
<code><source></code>	If <code><type></code> is <code>M-file</code> , <code>model</code> , <code>M-GUI</code> , then <code><source></code> is the name of the associated <code>.m</code> file or model file, with no extension. Otherwise, do not include a <code><source></code> element, but include a <code><callback></code> element.
<code><file></code>	Use this element only for examples with a <code><type></code> value other than <code>M-file</code> when you want to display an HTML file that describes the example. Specify a relative path from the location of <code>demos.xml</code> .

XML Tag	Notes
<callback>	Use this element only for examples with a <type> value of <code>video</code> or <code>other</code> to specify an executable file or a MATLAB command to run the example.
<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.

Define Categories Using <demosection> Tag

Optionally, define categories for your examples by including a <demosection> for each category. If you include *any* categories, then *all* examples must be in categories.

Each <demosection> element contains a <label> that provides the category name, and the associated <demoitem> elements.

See Also

`patchdemoxmlfile`

Related Examples

- “Display Custom Documentation” on page 31-21

Projects

- “Create Projects” on page 32-2
- “Automate Startup and Shutdown Tasks” on page 32-8
- “Manage Project Files” on page 32-10
- “Find Project Files” on page 32-12
- “Create Shortcuts to Frequent Tasks” on page 32-14
- “Add Labels to Project Files” on page 32-16
- “Create Custom Tasks” on page 32-18
- “Componentize Large Projects” on page 32-20
- “Share Projects” on page 32-23
- “Upgrade Projects” on page 32-27
- “Analyze Project Dependencies” on page 32-30
- “Clone from Git Repository” on page 32-44
- “Use Source Control with Projects” on page 32-45
- “Create and Edit Projects Programmatically” on page 32-56
- “Explore an Example Project” on page 32-63

Create Projects

What Are Projects?

A project is a scalable environment where you can manage MATLAB files, data files, requirements, reports, spreadsheets, tests, and generated files together in one place.

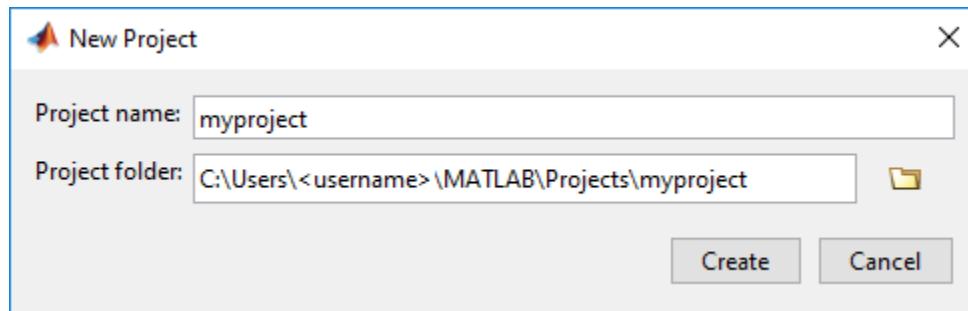
Projects can help you organize your work and collaborate. Projects promote productivity and teamwork by helping you with common tasks.

- Find all the files that belong with your project.
- Create standard ways to set up and shut down the MATLAB environment across a team.
- Create, store, and easily access common operations.
- View and label modified files for peer review workflows.
- Share projects using built-in integration with Git™, Subversion® (SVN), or using external source control tools.

Create Project

To create a blank project, on the **Home** tab, click **New > Project > Blank Project**. To create a project from an existing folder, on the **Home** tab, click **New > Project > From Folder**.

The New Project dialog box opens. Enter a project name, select a project folder, and click **Create**.



Open Project

To open an existing project, on the **Home** tab, click **Open** and browse to an existing project .prj file. Alternatively, in the Current Folder browser, double-click the project .prj file.

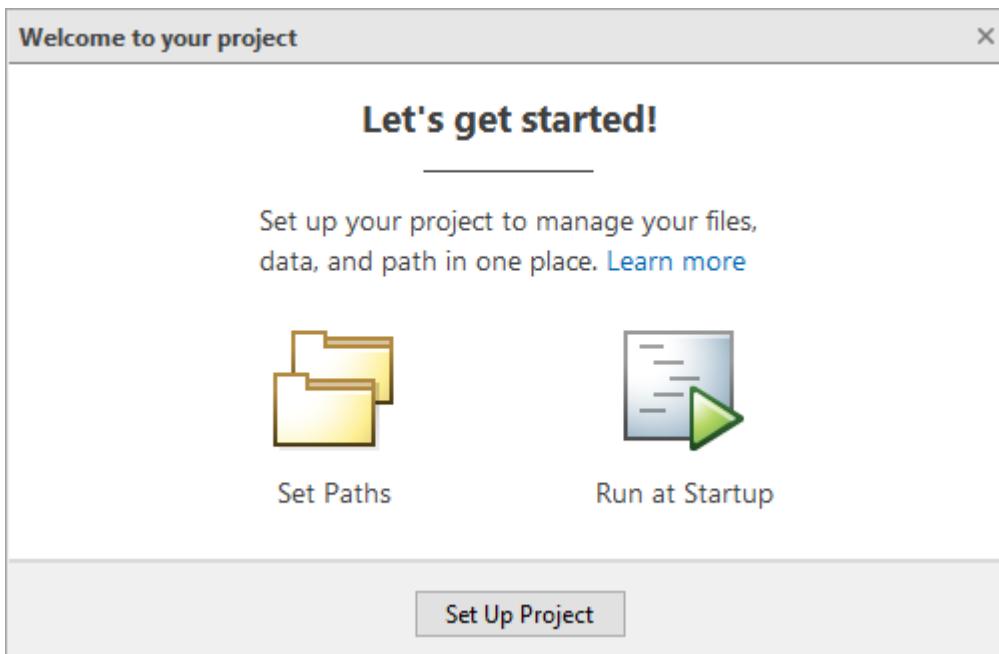
Note To avoid conflicts, you can have only one project open at a time. If you open another project, any currently open project closes.

To open a recent project, on the **Home** tab, click the **Open** arrow and select your project under the **Recent Projects** list.

Set up Project

After you create a project, the Welcome to your project dialog box opens and prompts you to set up the project.

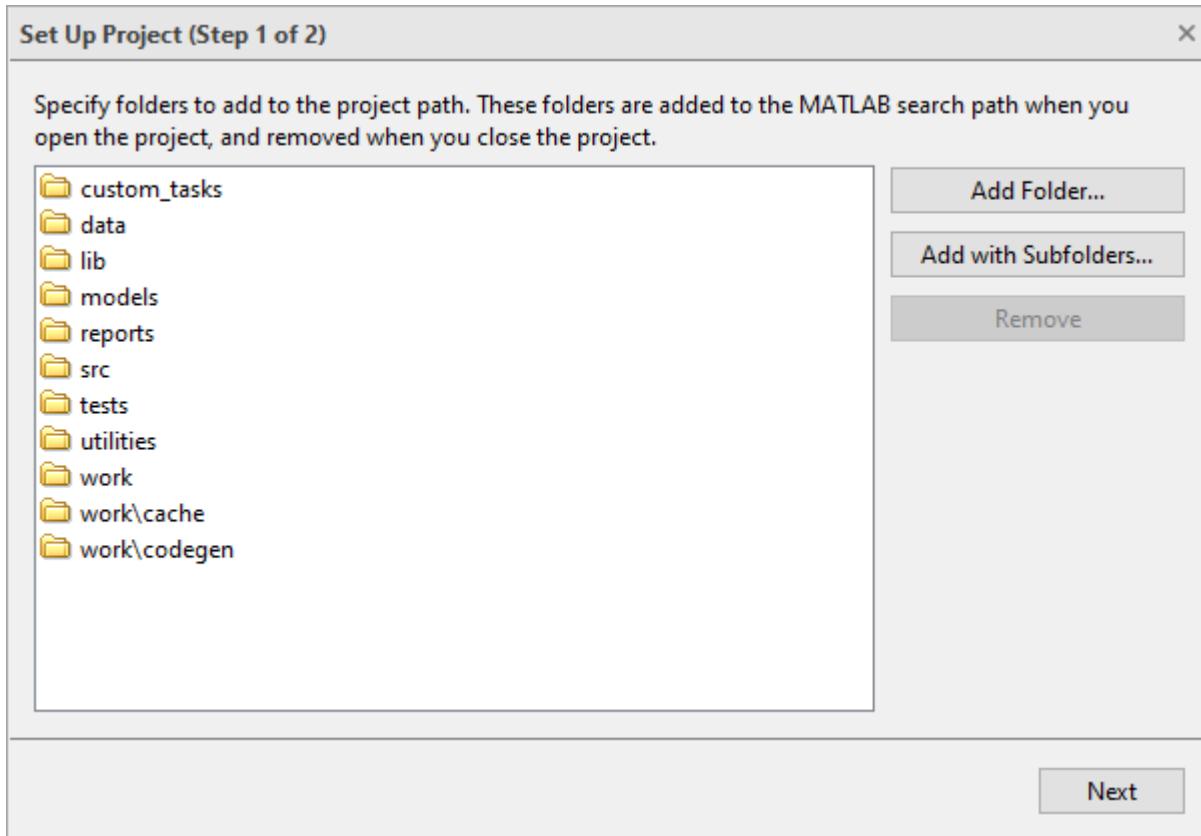
- 1 Click **Set Up Project** to begin setting up your project.



- 2 In the Set Up Project (Step 1 of 2) dialog box, you can choose folders to add to the project path. Adding project folders to the project path ensures that all users of the project can access the files within them. MATLAB adds these folders to the search path when you open the project, and removes them when you close the project.

To add all the folders in project folder to the project path, click **Add with Subfolders** and then select the root project folder containing all your subfolders.

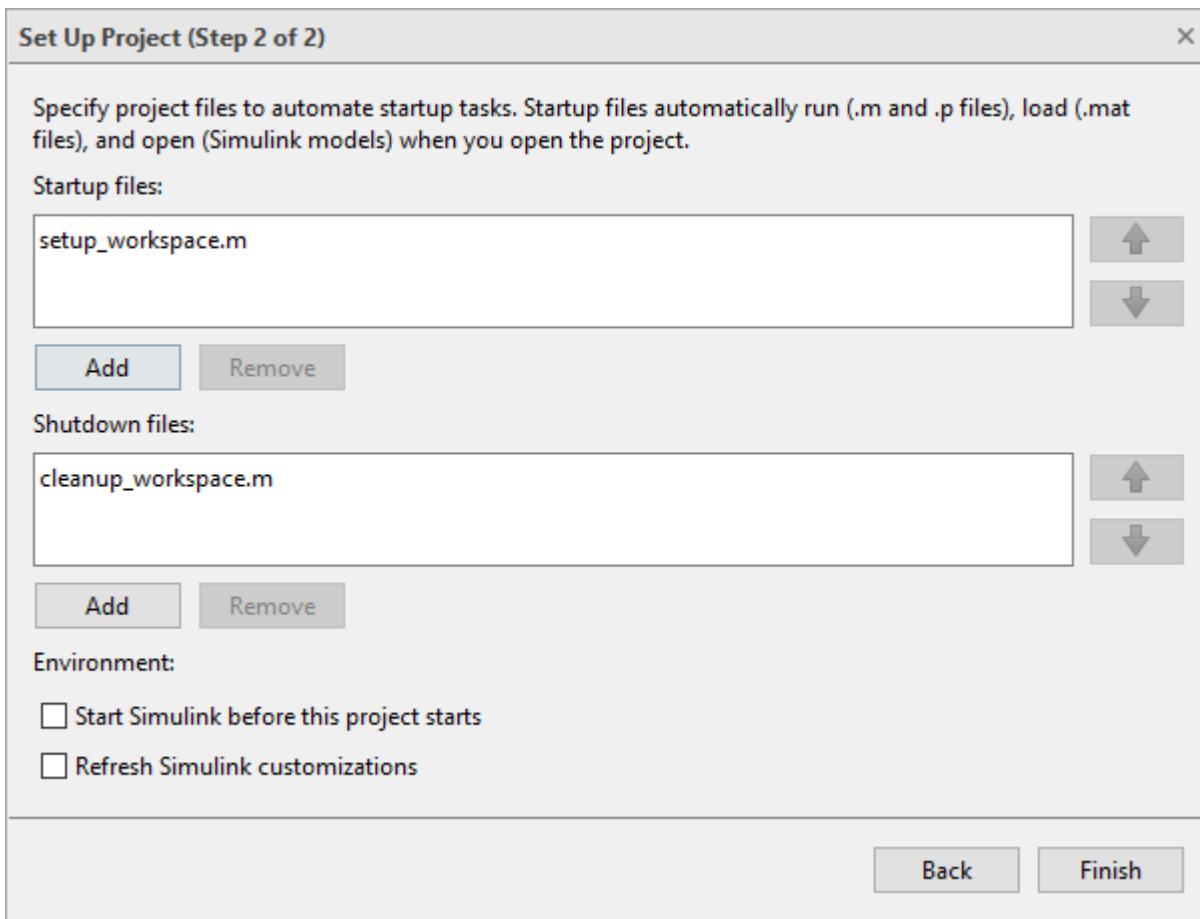
For more information about adding folders to the project path, including how to add them after completing the project setup, see "Specify Project Path" on page 32-8.



- 3 After specifying the project path, click the **Next** button to continue.
- 4 In the Set Up Project (Step 2 of 2) dialog box, you can specify startup and shutdown files. Startup files help you set up the environment for your project. Shutdown files help you clean up the environment when you are done. Use shutdown files to undo the setup that occurs in startup files.

Use the **Add** and **Remove** buttons to manage the startup and shutdown file lists. The files run from the top down. If the order in which the files are run is important, use the arrow buttons to move files up or down in the list.

For more information about specifying startup and shutdown files, including how to specify them after completing the project setup, see “Specify Startup and Shutdown Files” on page 32-8.



- 5 Click **Finish** to complete the project setup and open your new project.

Add Files to Project

The **Files** view shows the list of files in the project. Notice that, after creating a new project, the list is empty. Files in the project root folder are not included in the project until you add them.

To create a new file or folder in the project, in the **Files** view, right-click in white space and select **New Folder** or **New File**. The new file or folder is created and added to the project.

To add existing files to a project, on the **Project** tab, in the **Tools** section, select **Run Checks > Add Files**. Then, select from the list of files in the project folder that are not yet added to the project. To add existing files from the **Files** view, click **All**. Then, right-click one or more files and folders, and select **Add to Project** or **Add Folder to Project (Including Child Files)**. To add existing files from a file browser or the Current Folder browser, cut and paste or drag and drop the files into the project **Files** view. If you drag a file from outside the project root folder, this moves the file and adds it to your project. Drag files within the project root to move them.

To add and remove project files programmatically, use the `addFile` function.

You might not want to include all files in your project. For example, you might want to exclude some files in the project root folder, such as SVN or CVS source control folders. To determine which files need to be included in your project, see "Analyze Project Dependencies" on page 32-30.

Other Ways to Create Projects

There are several alternative ways to create a project. You can:

- Create a project from an archived project.
- Create a project using a Simulink template.
- Create a project using files retrieved from an existing source control repository. For more information on this last alternative, see “Use Source Control with Projects” on page 32-45.

Create a Project from an Archived Project

Some projects are shared as archived projects. An archived project is useful for sharing with users who do not have access to a connected source control tool. To view and edit the contents of an archived project, create a new project from the archived project.

To create a new project from an archived project, in the Current Folder browser, double-click the archived project file which has an `.mlproj` extension. The Extract Project dialog box opens. Specify the location for the new project and click **Select Folder**. For example, `C:\myNewProject`.

The new project opens. The current folder (for example, `C:\myNewProject`) contains the imported project folders. If the archived project contains referenced projects, MATLAB imports files into two subfolders, `mains` and `refs`. The `mains` project folder (for example, `C:\myNewProject\mains`) contains the project folders. The `refs` folder (for example `C:\myNewProject\refs`) contains the referenced project folders.

Create a Project Using Simulink

If you have Simulink, you can use a Simulink template to create and reuse a standard project structure.

To create a project from a Simulink template created in R2014b or later:

- 1 On the **Home** or **Project** tab, click **New > Project > From Simulink Template**. The Simulink Start Page opens.
- 2 The start page shows all project templates (`*.slt`) on the MATLAB path. Select a template in the list to read the template description.

If your templates do not appear, locate them by clicking **Open**. In the Open dialog box, make `*.slt` files visible by setting the file type to **All MATLAB files**, and browse to your template.

- 3 Select a template and click **Create Project**. The Create Project dialog box opens.

Templates created in R2017b or later warn you if required products are missing. Click the links to open Add-On Explorer and install required products.

- 4 Specify the root project folder, edit the project name, and click **Create Project**.

To use project templates created in R2014a or earlier (`.zip` files), upgrade them to `.slt` files using `Simulink.exportToTemplate`.

See Also

More About

- “Manage Project Files” on page 32-10
- “Analyze Project Dependencies” on page 32-30
- “Share Projects” on page 32-23
- “Use Source Control with Projects” on page 32-45
- “Create and Edit Projects Programmatically” on page 32-56

Automate Startup and Shutdown Tasks

When you open a project, MATLAB adds the project path to the MATLAB search path and then runs or loads specified startup files. The project path and startup files help you set up the environment for your project. Similarly, when you close a project, MATLAB removes the project path from the MATLAB search path and runs specified shutdown files. Shutdown files help you clean up the environment for your project. Use shutdown files to undo the setup that occurs in startup files.

More specifically, when you open a project, MATLAB changes the current folder to the project startup folder, and runs (.m and .p files) or loads (.mat files) any specified startup file. For more information about configuring the startup folder, see “Set Startup Folder” on page 32-8.

Specify Project Path

You can add or remove folders from the project path. Adding a project folder to the project path ensures that all users of the project can access the files within it.

To add a folder to the project path, on the **Project** tab, in the **Environment** section, click **Project Path**. Click **Add Folder** and select the folder that you want to add. To add a folder and all of its subfolders, click **Add with Subfolders**.

To remove a folder from the project path, select the folder from the displayed list and click **Remove**.

You also can add or remove a folder from the project **Files** view. Right-click the folder, select **Project Path**, and select from the available options.

Folders on the project path appear with the project path icon in the **Status** column of the **Files** view.

Set Startup Folder

When you open the project, the current working folder changes to the project startup folder. By default, the startup folder is set to the project root.

To edit the project startup folder, on the **Project** tab, in the **Environment** section, click **Details**. Then, in the **Start up** section, enter a path for the project startup folder.

Specify Startup and Shutdown Files

To configure a file to run when you open your project, right-click the file and select **Run at Startup**. To configure a file to run when you close your project, right-click the file and select **Run at Shutdown**. The **Status** column displays an icon indicating whether the file runs at startup or shutdown.

To stop a file from running at startup or shutdown, right-click the file and select **Remove from Startup** or **Remove from Shutdown**.

Alternatively, go to the **Project** tab and click **Startup Shutdown**. Then, in the Manage Project Startup and Shutdown dialog box, use the **Add** and **Remove** buttons to manage the startup and shutdown file lists. The files run from the top down. If the order in which the files are run is important, use the arrow buttons to move files up or down in the list.

Note Startup and shutdown files are included when you commit modified files to source control. When you configure startup and shutdown files, they run for all other project users.

Startup files can have any name except `startup.m`. A file named `startup.m` on the MATLAB path runs when you start MATLAB. If your `startup.m` file calls the project, an error occurs because the project is not yet loaded. For more information about using `startup.m` files, see “Startup Options in MATLAB Startup File”.

To create new startup and shutdown files programmatically, see `addStartupFile` and `addShutdownFile`.

In Simulink, you can specify additional project startup options. For more information, see “Automate Startup Tasks” (Simulink).

See Also

`addStartupFile`

More About

- “Manage Project Files” on page 32-10
- “Create and Edit Projects Programmatically” on page 32-56
- “What Is the MATLAB Search Path?”

Manage Project Files

This table shows how to add, move, rename, and open project files and folders. Some of these actions can also lead to automatic updates that affect other files. All can be undone and redone.

Action	Procedure
View project files.	In the Files view, click Project to display only the files and folders that are included in the project.
View all files in a project folder.	To display all files and folders in the project folder, in the Files view, click All . You might not want to include all files in your project. For example, you might want to exclude SVN or CVS source control folders. For more information, see “Work with Derived Files in Projects” on page 32-54.
Create a new project folder.	In the Files view, right-click in white space, and then click New > Folder .
Add files to a project.	On the Project tab, in the Tools section, select Run Checks > Add Files . Select from the list of unmanaged files in the project folder. You also can paste or drag files and folders from your operating system file browser or the Current Folder browser to the project Files view. When you drag a file to the Files view, MATLAB adds the file to the project. To add a file programmatically, use the <code>addFile</code> and <code>addFolderIncludingChildFiles</code> functions. For example, to add a file named <code>myfile.m</code> to the <code>proj</code> project object, type <code>addFile(proj, 'myfile.m');</code>
Remove project files or folders.	In the Files view, right-click the file, and then select Remove from Project . To remove a file programmatically, use the <code>removeFile</code> function.
Move project files or folders.	Cut and paste or drag the files in the project.
Rename project files or folders.	In the Files view, right-click the file, and then click Rename .
Open project files.	In the Files view, right-click the file, and then click Open . You also can double-click the file.
Preview project file contents without opening the file.	In the Files view, select the file. The panel in the bottom right of the Files view displays the file information and labels. To restore the panel if it is minimized, click the Restore  button.
Delete a project file or folder.	In the Files view, right-click the file, and then click Delete .

Action	Procedure
Undo or redo an action	<p>Click  or  at the upper right corner of the toolbar.</p> <p>If you are using source control, you also can revert to a particular version of a file or project. For more information, see "Revert Changes" on page 32-53.</p>

Automatic Updates When Renaming, Deleting, or Removing Files

When you rename, delete, or remove files or folders in a project, the project runs a dependency analysis to check for effects on other project files. When the analysis is complete, the project displays the affected files. If you have not yet run a dependency analysis on the project, the analysis may take some time to run. Subsequent analyses are incremental updates, and so they run faster.

When renaming files, the project offers to automatically update references to the file. Automatically updating references to a file when renaming prevents errors that result from changing names or paths manually and overlooking or mistyping the name.

For example:

- When renaming a class, the project offers to automatically update all classes that inherit from it.
- When renaming a .m or .mlx file, the project offers to automatically update any files and callbacks that call it.
- When renaming a C file, the project prompts you to update the S-function that uses it.

For more information about automatic updates when renaming, deleting, or removing Simulink files such as library links, model references, and model callbacks, see "Automatic Updates When Renaming, Deleting, or Removing Files" (Simulink).

See Also

More About

- "Find Project Files" on page 32-12
- "Add Labels to Project Files" on page 32-16
- "Create Custom Tasks" on page 32-18

Find Project Files

There are several ways to find files and folders in projects. You can:

- Group and sort project files.
- Search for and filter project files.
- Search the content in project files.

Group and Sort Project Files

To change how files and folder are grouped or sorted in the **Files** view, select from the options in the **Layout** field and the Actions menu .

For example, to group files by type, in the **Layout** field, select **List**. Then, click the Actions button  and select **Group By > Type**. To sort the grouped files by size, click the actions  button and select **Sort By > Size**.

Search for and Filter Project Files

You can search for project files and folders in the **Files** view.

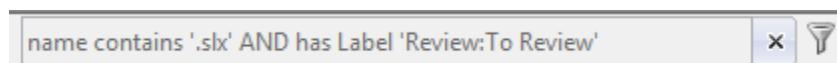
To search for a file or folder, in the search field at the top of the view, begin typing the file or folder name. The asterisk character (*) is a wildcard. For example, to show only file names that begin with **coll** and have a **.m** extension, type **coll*.m**.

To filter the files and folders in the current view, click the Filter button  next to the search field. In the Filter Builder dialog box, select the names, file types, project statuses, and labels to filter by.

The Filter Builder displays the resulting filter. For example:

```
Filter = type=='MATLAB Code files (*.m, *.mlx)' AND project status=='In project'  
AND label=='Classification:Test'
```

Click **Apply** to filter the files and folders in the current view. The search field displays the applied filter.



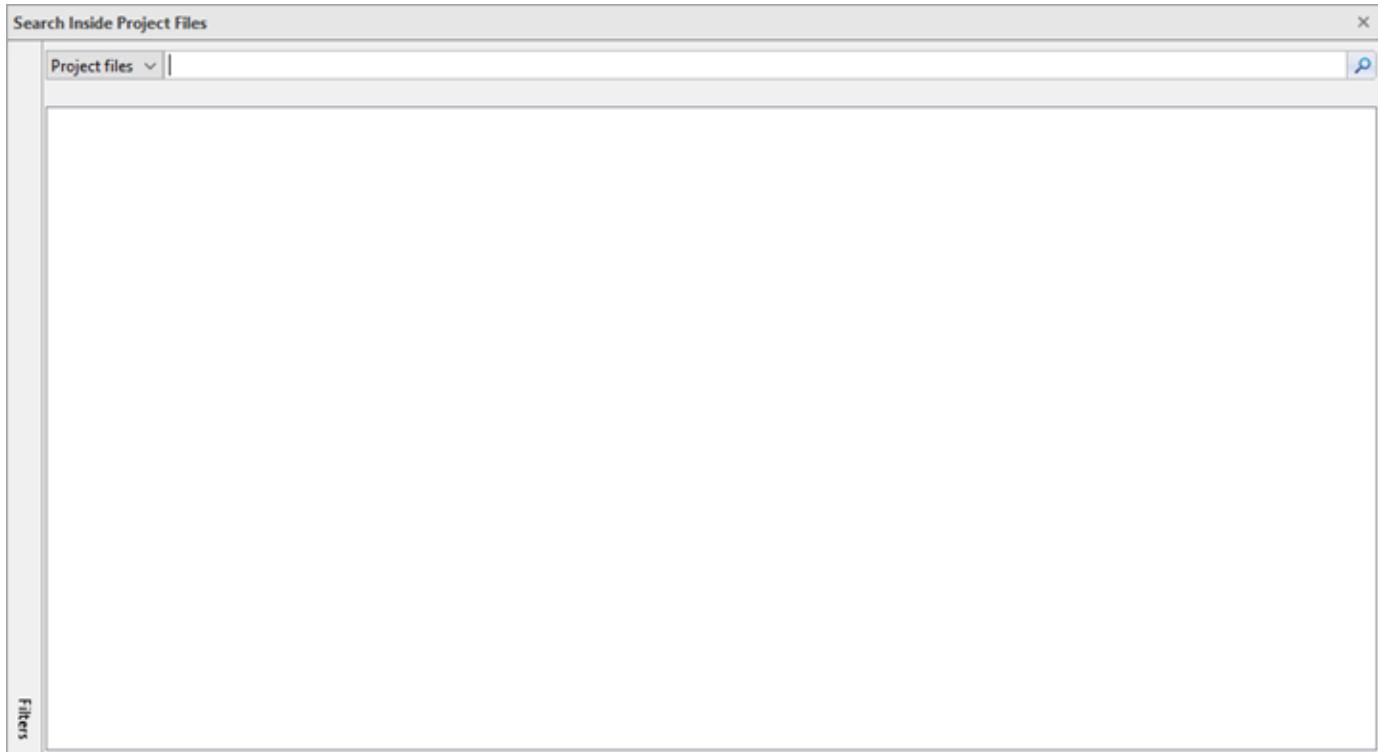
To clear a search or filter, click the Clear button  to the right of the search field.

Search the Content in Project Files

You can search for text inside MathWorks product files (such as MATLAB files, Simulink model files, and data dictionaries) and some other kinds of project files (such as PDF and Microsoft Word files). MATLAB only searches in files in the project. To search the content of files in a referenced project, open the referenced project.

To search the content of project files:

- 1 On the **Project** tab, click **Search**. The Search Inside Project Files dialog box opens.



- 2 Enter the text to search. MATLAB searches the project files for the exact text in the search field and displays the results.

Avoid using quotes around phrases.

- 3 Expand a result in the result list to view the match and surrounding context inside the search window.
- 4 Double-click the search result to open the file and locate the match.
- 5 Click **Filters** at the bottom left of the results to refine the results by file type, status, or label.

See Also

More About

- “Manage Project Files” on page 32-10
- “Create Shortcuts to Frequent Tasks” on page 32-14
- “Add Labels to Project Files” on page 32-16

Create Shortcuts to Frequent Tasks

You can create shortcuts in projects to perform common project tasks, such as opening important files and loading data.

Run Shortcuts

To run a shortcut, in the **Project Shortcuts** tab, click the shortcut. Clicking a shortcut in the **Project Shortcuts** tab performs the default action for the file type. For example, MATLAB runs .m shortcut files and loads .mat shortcut files. If the shortcut file is not on the path, MATLAB changes the current folder to the parent folder of the shortcut file, runs the shortcut, and then changes the current folder back to the original folder.

Alternatively, in the **Files** view, you can right-click the shortcut file and select **Run**. If the script is not on the path, then MATLAB asks if you want to change folder or add the folder to the path.

Create Shortcuts

To create a shortcut from an existing project file:

- 1 In the **Files** view, right-click the file and select **Create Shortcut**. Alternatively, on the **Project Shortcuts** tab, click **New Shortcut** and browse to select a file.
The Create New Shortcut dialog box opens.
- 2 Select an icon and enter a name. If using shortcuts to define steps in a workflow, consider prefixing their names with numbers.
- 3 To add the shortcut to an existing group, in the **Group** field, select a group from the drop-down list. For more information about shortcut groups, see “Organize Shortcuts” on page 32-14.
- 4 Click **OK**.

The shortcut appears with the selected name and icon on the **Project Shortcuts** tab. In the **Files** view, the **Status** column displays an icon indicating that the file is a shortcut.

Note Shortcuts are included when you commit your modified files to source control, so you can share shortcuts with other project users.

Organize Shortcuts

You can organize shortcuts by categorizing them into groups. For example, you might create one group of shortcuts for loading data, another for opening files, another for generating code, and another for running tests.

To create a shortcut group:

- 1 On the **Project Shortcuts** tab, click **Organize Groups**.
- 2 Click the **Create** button.
- 3 Enter a name for the group and click **OK**.

The new shortcut group appears on the **Project Shortcuts** tab.

To move a shortcut into a group:

- 1 On the **Project Shortcuts** tab, right-click a shortcut and select **Edit Shortcut**. Alternatively, in the **Files** view, right-click a file and select **Edit Shortcut**.

The Create New Shortcut dialog box opens.

- 2 In the **Group** field, select a group from the drop-down list and click **OK**.

Shortcuts are organized by group in the **Project Shortcuts** tab.

See Also

More About

- “Manage Project Files” on page 32-10
- “Find Project Files” on page 32-12
- “Add Labels to Project Files” on page 32-16

Add Labels to Project Files

You can use labels to organize project files and communicate information to project users.

Add Labels

To add a label to a project file, in the **Files** view, select the file. Then, drag the desired label from the **Labels** panel at the bottom left of the project into the Label Editor panel for the selected file. The Label Editor panel is located at the bottom right of the **Files** view. To restore the panel if it is minimized, click the  icon.

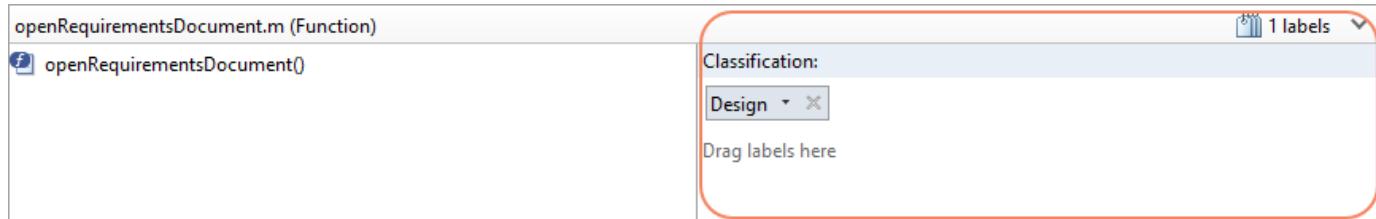
To add a label to multiple project files, in the **Files** view or in the Dependency Analyzer graph, select the files, right-click, and select **Add Label**. Choose a label from the list and click **OK**.

Note After you add a label to a file, the label persists across file revisions.

To add labels programmatically (for example, in custom task functions) see `addLabel`.

View and Edit Label Data

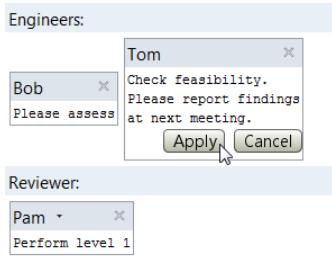
When you select a project file in the **Files** view, the file labels appear in the Label Editor view.



To change a label that belongs to a single-valued category, select the new value from the label list.



You can add additional annotations to labels from categories that you create. In the Label Editor panel, click a label and insert or modify text. Then, click **Apply**.



Create Labels

Labels exist in two types of categories:

- Single-valued — You can attach only one label from the category to a file.
- Multi-valued — You can attach multiple labels from the category to a file.

All projects contain a built-in label category called **Classification** with several built-in labels. These built-in labels are read-only.

To create your own label categories:

- 1 In the **Labels** panel at the bottom left of the project, right-click and select **Create New Category**. The Create Category dialog box opens.
- 2 Enter a name for the new category.
- 3 To create a single-valued label category, select the **Single Valued** check box. Otherwise, MATLAB creates a multi-valued label category.
- 4 To specify a label data type other than the default **String** data type, from the **Type** list, select from the available options.
- 5 Click **Create**.

To create your own labels in a label category:

- 1 In the **Labels** panel at the bottom left of the project, right-click the label category and select **Create New Label**. The Create Label dialog box opens.
- 2 Enter a name for the new label and click **OK**.

To rename or delete a category or label, right-click it and select **Rename** or **Remove**.

To create a new label or label category programmatically, see `createLabel` or `createCategory`.

See Also

More About

- “Manage Project Files” on page 32-10
- “Find Project Files” on page 32-12

Create Custom Tasks

Custom tasks are MATLAB functions that allow you to perform a series of operations on one or more files. You can create a custom task function and then run the custom task on a select set of files in your project. For example, you can create a custom task to check all the code files for errors or to run all the tests in the project.

Create a Custom Task Function

To create a custom task function:

- 1 In the **Project** tab, click **Custom Tasks** and then select **Manage Custom Tasks**. The Manage Custom Tasks dialog box opens.
- 2 Click **Add** and then select **Add Using New Function**. If you want to add an existing script as a custom task, select **Add Using Existing Function**.
- 3 Specify a file name for the script and save the new file on the MATLAB path. The MATLAB Editor opens the new file containing an example custom task function.
- 4 Edit the function to perform the desired action on each file. Use the instructions at the top of the file to guide you to create a custom task with the correct function signature. Your custom tasks must accept a full path to a file as the single input argument and return a single output argument.

For example, this custom task function extracts Code Analyzer information for each file using the `checkcode` function.

```
[~,~,ext] = fileparts(file);
switch ext
    case {'.m', '.mlx', '.mlapp'}
        result = checkcode(file, '-string');
    otherwise
        result = [];
end
```

- 5 Save the file.

You can use the MATLAB editor to set breakpoints and debug a custom task function, just as with any other MATLAB function.

Run a Custom Task

To run a custom task on a select set of files in your project:

- 1 On the **Project** tab, click **Custom Tasks** and then select **Run Custom Task**.
- 2 In the **Include** column of the table, select which project files you want to run the custom task on.

To include or exclude multiple files in the table at once, press the **Shift** or **Ctrl** key, select the files, and then right-click and select **Include** or **Exclude**. If the custom task function can identify the files to operate on, include all files.

- 3 In the **Custom task** field, select from the available custom task functions. You also can enter the name of a task directly in the field, or click **Browse**.
- 4 Click **Run Task** to run the task. The Custom Task Report window displays the results.

- 5 Examine the **Results** column in the table to ensure that the custom task ran correctly on all files. To view detailed result information for a file, select the file in the table. The results pane at the bottom of the Custom Task Report displays the details.

Save Custom Task Report

Saving a Custom Task Report is useful if you want to save a record of the custom task results, or if you want to share the results with others.

To save the Custom Task Report, click the **Publish Report** button at the bottom of the Custom Task Report. You can either save the report as an HTML file or a Microsoft Word file. If you have MATLAB Report Generator™, you also can save the report as a PDF file.

To see the report file and add it to your project, switch to the **All** files view.

See Also

More About

- “Create Shortcuts to Frequent Tasks” on page 32-14
- “Manage Project Files” on page 32-10

Componentize Large Projects

MATLAB supports large-scale project componentization by allowing you to reference other projects from a parent project. Organizing large projects into components facilitates code reuse, modular and team-based development, unit testing, and independent release of components.

Projects can reference multiple other projects in a hierarchical manner. The project reference hierarchy appears as a tree in the **References** view.

From a parent project, you can

- Access the project paths, entry-point shortcuts, and source control information for all referenced projects.
- View, edit, and run files that belong to a referenced project.
- Detect changes in referenced projects using checkpoints.

Add or Remove Reference to a Project

You can add new components to a project by referencing other projects.

To add a reference to a project:

1

On the **Project** tab, in the **Environment** section, click  **References**. The Add Reference dialog box opens.

2 Browse to select the required project (.prj) file.

3 In the **Reference type** field, select either **Relative** or **Absolute**. Select **Relative** if your project hierarchy has a well-defined root relative to your project root. For example, your project root might be a folder under source control. Select **Absolute** if the project you want to reference is in a location accessible to your computer, for example, a network drive.

4 To create a checkpoint when you add the project, select **Set a checkpoint to detect future changes**. For more information about checkpoints, see “Manage Changes in Referenced Project Using Checkpoints” on page 32-21.

5 Click **Add**.

When the referenced project loads, MATLAB adds the referenced project path to the MATLAB search path and then runs or loads specified startup files. Similarly, when the referenced project closes, MATLAB removes the project path from the search path and runs specified shutdown files. MATLAB loads referenced projects before their parent projects. This allows the parent project to access the referenced project in startup and shutdown files.

To remove a referenced project from your project hierarchy, in the **References** tree, right-click the referenced project and select **Remove Reference**.

View, Edit, or Run Referenced Project Files

If you have a project that references other projects, you can view, modify, or run the files that belong to the referenced projects directly from the parent project.

To view a referenced project, in the parent project, select the **References** view. In the **References** tree, select a referenced project.

To display the referenced project files, at the top right of the **References** view, click **Show Files**.

To modify or run a file, right-click the file and select from the list of available options.

Extract Folder to Create a Referenced Project

You can extract an existing folder in a project to create a referenced project. After extracting a folder, file and folder contents and shortcuts in the referenced project remain accessible from the parent project.

To extract a folder from a project and convert the folder into a referenced project:

- 1** In the **Files** view, right-click the folder and select **Extract to Referenced Project**. The Extract Folder to New Project dialog box opens.
- 2** Specify a project name and location
- 3** In the **Reference type** field, select either **Relative** or **Absolute**. Select **Relative** if you specify the new project location with reference to the current project root. Select **Absolute** if you specify the full path for the new location, which is, for example, on a network drive
- 4** To disable any of the default content migration actions, click **More Options** and clear the corresponding check boxes.
- 5** Click **Extract**.
- 6** In the two Warning dialog boxes that open, click **OK**.

The selected folder and its contents are removed from the project. On the **Project Shortcuts** tab, the **Referenced Projects** section shows a new shortcut for the referenced project.

Manage Changes in Referenced Project Using Checkpoints

To detect and compare changes in a referenced project, create checkpoints. You then can compare the referenced project against the checkpoint to detect changes.

By default, MATLAB creates a checkpoint when you add a reference to a project. To create additional checkpoints:

- 1** In the parent of the referenced project, select the **References** view.
- 2** To create a checkpoint, go to the **References** tab, and in the **Checkpoint** section, click  **Update**. In the **Details** view, the **Checkpoint** field displays the timestamp of the latest checkpoint.

To detect changes in a referenced project, go to the **References** tab, and in the **Checkpoint** section, click  **Checkpoint Report**. The Difference to Checkpoint dialog box displays the files that have changed on disk since you created the checkpoint.

To remove the checkpoint, in the **Checkpoint** section of the **References** tab, click  **Clear**.

See Also

More About

- “Create Projects” on page 32-2
- “Share Projects” on page 32-23

Share Projects

You can collaborate with others by packaging and sharing projects. You can also control which files are included in a shared project by using labels and an export profile. (For more information, see “Create an Export Profile” on page 32-26.)

This table describes the ways of packaging a project.

Way to Share Projects	Procedure
Create an archive to share.	<p>You can convert your project to a .mlproj or ZIP archive and share the archive with collaborators. Sharing an archive is useful when working with someone who does not have access to a connected source control tool.</p> <p>To archive a project:</p> <ol style="list-style-type: none">1 With the project loaded, on the Project tab, click Share > Archive.2 If you want to export only a set of specified files, choose an Export profile. For more information, see “Create an Export Profile” on page 32-26.3 If you have referenced projects and want to export the referenced project files, select Include referenced projects.4 Click Save As and specify a file path.5 In the Save as type field, select the archived project file type. By default, MATLAB archives the project as an .mlproj file. You can choose to archive the project as a ZIP file.6 Click Save to create the project archive. <p>You can now share the archive file the way you would any other file.</p>

Way to Share Projects	Procedure
Email an .mlproj archive in one step.	<p>On Windows systems, you can generate an .mlproj archive and attach it to an email in one step. Sharing an archive is useful when working with someone who does not have access to a connected source control tool.</p> <p>To email a project as an .mlproj file:</p> <ol style="list-style-type: none"> With the project loaded, on the Project tab, select Share > Email. If you want to export only a set of specified files, choose an Export profile. For more information, see “Create an Export Profile” on page 32-26. If you have referenced projects and want to export the referenced project files, select the Include referenced projects check box. Click Attach to Email. MATLAB opens a new email in your default email client with the project attached as an .mlproj file. Edit and send the email.
Create a toolbox to share.	<p>You can create a toolbox from your project and share the toolbox with collaborators.</p> <p>To create a toolbox from your project:</p> <ol style="list-style-type: none"> With a project loaded, on the Project tab, select Share > Toolbox. The packager adds all project files to the toolbox and opens the Package a Toolbox dialog box. The Toolbox Information fields are populated with the project name, author, and description. Edit the information as needed. To include files not already included in the project files, edit the excluded files and folders. Click Package. <p>You can now share the toolbox file the way you would any other file.</p>

Way to Share Projects	Procedure
Publish on GitHub®.	<p>You can share a project by making it publicly available on GitHub. You must first have a GitHub account.</p> <p>Sharing a project on GitHub adds Git source control to the project. If your project is already under source control, sharing replaces the source control configuration with Git, and GitHub becomes the project's remote repository.</p> <p>Note If you do not want to change your current source control in the open project, share the project in another way.</p> <p>To share a project on GitHub:</p> <ol style="list-style-type: none"> With a project loaded, on the Project tab, select Share > GitHub. The Create GitHub Repository dialog box opens. If GitHub is not an option in the Share menu, select Share > Change Share Options. Then, in the Manage Sharing dialog box, select GitHub and click Close. Enter your GitHub user name and personal access token, and edit the name for the new repository. Click Create. <p>A warning prompts you to confirm that you want to create a public repository and modify the current project's remote repository location. To continue, click Yes.</p> <ol style="list-style-type: none"> The Create GitHub Repository dialog box displays the URL for the new repository. Click the link to view the new repository on the GitHub website. The repository contains the initial check-in of your project files. The source control in your current project now references the new repository on GitHub as the remote repository. To use the project with the new repository, in the Create GitHub Repository dialog box, click Reload Project. <p>To view the URL for the remote repository, on the Project tab, in the Source Control section, click the Git Details button.</p> <p>You can use Git without any additional installation. To perform merging operations in Git, additional setup steps are required. For more information, see "Set Up Git Source Control" on page 33-23.</p>
Collaborate using source control.	You can collaborate using source control within projects. For more information, see "Use Source Control with Projects" on page 32-45.

Before sharing a project with others, it can be useful to examine the required add-ons for your project by performing a dependency analysis. For more information, see "Find Required Products and Add-Ons" on page 32-40.

Create an Export Profile

If you want to exclude files from a project before sharing it, create an export profile. An export profile allows you to exclude files with particular labels. For more information about creating labels and adding them to project files, see “Create Labels” on page 32-17.

To create an export profile:

- 1 On the **Project** tab, click **Share > Manage Export Profiles**.
- 2 Click **+** and specify a name for the export profile.
- 3 In the **Files** pane, click **+** and select the labels for the files you do not want to export, and then click **OK**.
- 4 In the **Labels** pane click **+** and select the custom labels you do not want to export, and then click **OK**.
- 5 Click **Apply** and close the Manage Export Profiles dialog box.

Note Export profiles do not apply changes to referenced projects. When you share your project, MATLAB exports the entire referenced projects.

See Also

More About

- “Add Labels to Project Files” on page 32-16
- “Create Projects” on page 32-2
- “Analyze Project Dependencies” on page 32-30

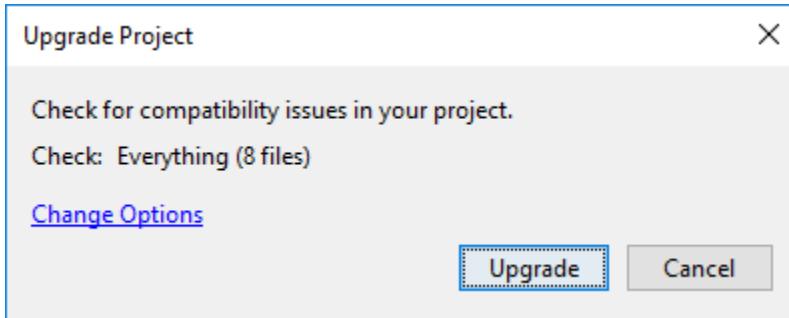
Upgrade Projects

The Upgrade Project tool helps you check for compatibility issues or upgrade your project to the current MATLAB release. The tool applies fixes automatically when possible and produces a report.

Tip To perform an upgrade that you can later revert if necessary, add source control to your project before upgrading. For more information, see “Use Source Control with Projects” on page 32-45.

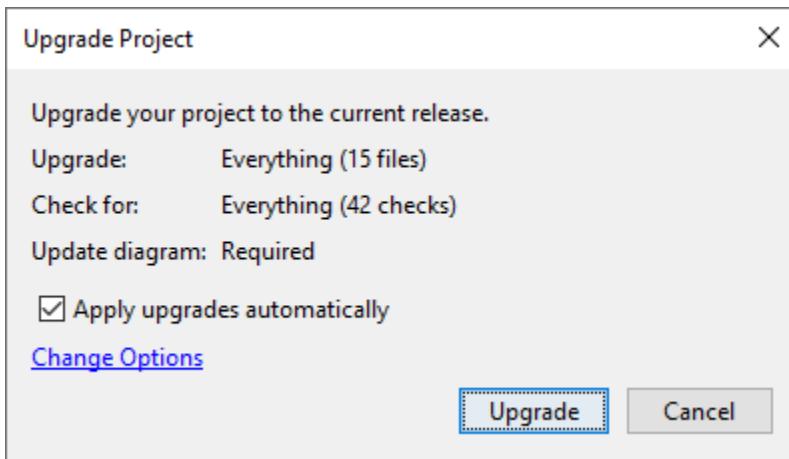
Run Upgrade Project Tool

- For projects that contain only MATLAB files, you can check for compatibility issues with the current MATLAB release. On the **Project** tab, click **Run Checks > Upgrade**.



Click **Upgrade** to check for compatibility issues.

- For projects that also contain Simulink models and libraries, you can apply fixes and automatically upgrade your project to the current release. On the **Project** tab, click **Run Checks > Upgrade**.



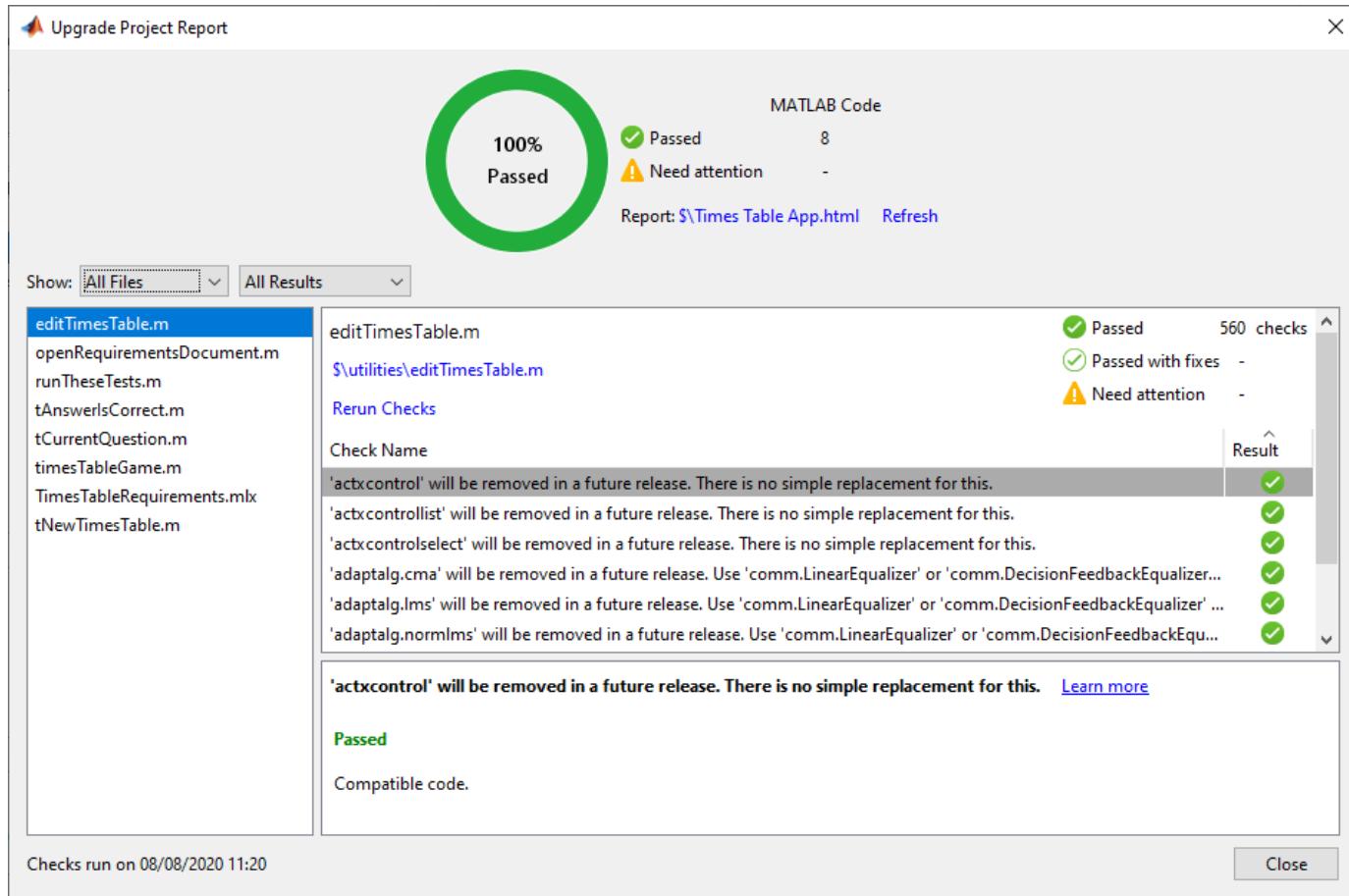
If you want to run the Project Upgrade tool without applying fixes automatically, clear **Apply upgrades automatically**.

If you want to specify which files to upgrade and which checks to run, click **Change Options**. In the Upgrade Options dialog box, clear the check box for any file or check that you want to exclude from the upgrade. For example, you might want to exclude checks that require performing an update diagram, as this can be time consuming.

The Upgrade Project tool displays the results of the compatibility check or upgrade in the Upgrade Project Report.

Examine Upgrade Project Report

After upgrading a project, examine the Upgrade Project Report to ensure that the upgrade occurred as planned.



The summary shows how many files passed all of the upgrade checks and how many files require attention. To view the upgrade results for a file, select the file in the left file list. By default, the file list shows any files that need attention. To change which files are shown, select from the options in the **Show** drop-down.

For each file in the Project Upgrade Report, examine the checks marked as needing attention. These files are marked with an orange circle icon in the **Result** column. Select a check in the **Check Name** column to display the check results and any automatically applied fixes in the lower panel.

To see the differences before and after the upgrade, in the Upgrade Project Report, click **View Changes**. If your project is under source control, you also can see the differences by comparing the before and after versions of the file.

MATLAB saves an HTML report of the upgrade results in the project root folder. To open the saved report, click the **Report** link at the top of the Upgrade Project Report.

See Also

More About

- “Use Source Control with Projects” on page 32-45
- “Analyze Project Dependencies” on page 32-30

Analyze Project Dependencies

Use the Dependency Analyzer to perform a dependency analysis on your project. You can run a dependency analysis at any point in your workflow. In a collaborative environment, you typically check dependencies:

- When you set up or explore a project for the first time
- When you run tests to validate changes to your design
- Before you submit a version of your project to source control
- Before you share or package your project

To explore a project and visualize its structure using different views, see “Explore the Dependency Graph, Views, and Filters” on page 32-32.

To find and fix problems in your project, see “Investigate and Resolve Problems” on page 32-37.

To assess how a change will affect other project files, see “Find File Dependencies” on page 32-41.

To find add-ons and products required by your project to run properly, see “Find Required Products and Add-Ons” on page 32-40.

Run a Dependency Analysis

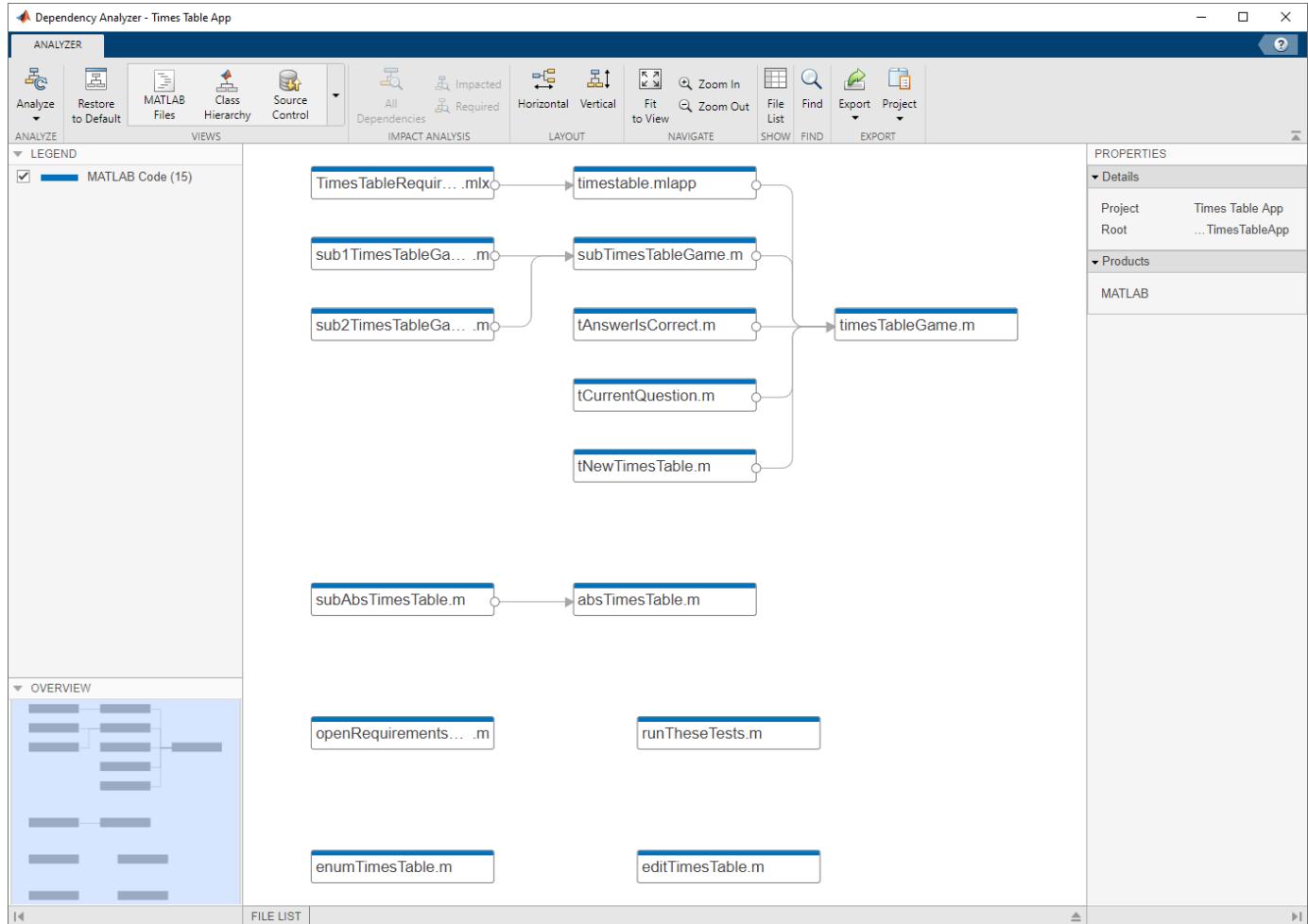
Before running a dependency analysis on a project, make sure that you have added all your files to the project. For more information, see “Add Files to Project” on page 32-5.

To start analyzing your project, on the **Project** tab, in the **Tools** section, click **Dependency Analyzer**. Alternatively, in the project **Views** pane, select **Dependency Analyzer** and click **Analyze**.

To analyze the dependencies of specific files, in the dependency graph, select the files. In the **Impact Analysis** section, click **All Dependencies** or use the context menu and select **Find All Dependencies**.

To analyze the dependencies inside add-ons, select **Analyze > Add-Ons**. For more details about available options, see “Analysis Scope” (Simulink).

You can also check dependencies directly in Project. In the Project **Files** view, right-click the project files you want to analyze and select **Find Dependencies**.



The dependency graph shows:

- Your project structure and its file dependencies, including how files such as models, libraries, functions, data files, source files, and derived files relate to each other.
- Required products and add-ons.
- Relationships between source and derived files (such as .m and .p files, .slx and .slxp, .ssc and .sscp, or .c and .mex files), and between C/C++ source and header files. You can see what code is generated by each model, and find what code needs to be regenerated if you modify a model.
- Warnings about problem files, such as missing files, files not in the project, files with unsaved changes, and out-of-date derived files.

You can examine project dependencies and problem files using the **File List**. In the toolbar, click **File List**.

After you run the first dependency analysis of your project, subsequent analyses incrementally update the results. The Dependency Analyzer determines which files changed since the last analysis and updates the dependency data for those files. However, if you update add-ons or installed products and want to discover dependency changes in them, you must perform a complete analysis. To perform a complete analysis, in the Dependency Analyzer, click **Analyze > Reanalyze All**.

For more information about running a dependency analysis on Simulink models and libraries, see “Perform an Impact Analysis” (Simulink).

Explore the Dependency Graph, Views, and Filters

The dependency graph displays your project structure, dependencies, and how files relate to each other. Each item in the graph represents a file and each arrow represents a dependency. For more details, see “Investigate Dependency Between Two Files” on page 32-32.

By default, the dependency graph shows all files required by your project. To help you investigate dependencies or a specific problem, you can simplify the graph using one of the following filters:

- Use the filtered **Views** to color the files in the graph by type, class, source control status, and label. See “Color Files by Type, Status, or Label” on page 32-33.
- Use the check boxes in the **Legend** pane to filter out a group of files.
- Use the **Impact Analysis** tools to simplify the graph. See “Find File Dependencies” on page 32-41.

Select, Pan, and Zoom

- To select an item in the graph, click it.

To select multiple files, press **Shift** and click the files.

To select all files of a certain type, hover the pointer over the corresponding item in the **Legend** pane and click the **Add to selection** icon.

To clear all selection, click the graph background.

To remove all files of a certain type from the current selection, hover the pointer over the corresponding item in the **Legend** pane and click the **Remove from selection** icon.

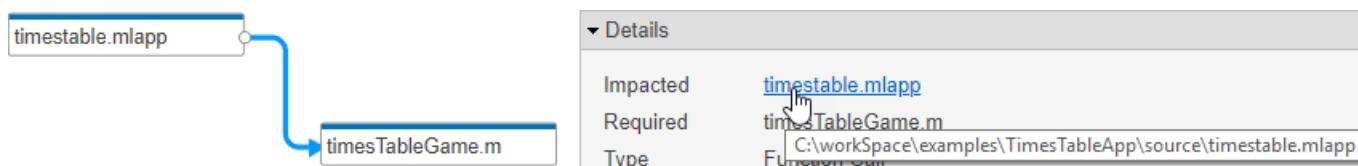
- To open a file, double-click it.
- To pan the dependency graph, hold the **Space** key, click and drag the mouse. Alternatively, press and hold the mouse wheel and drag.

For large graphs, navigate using the **Overview** pane.

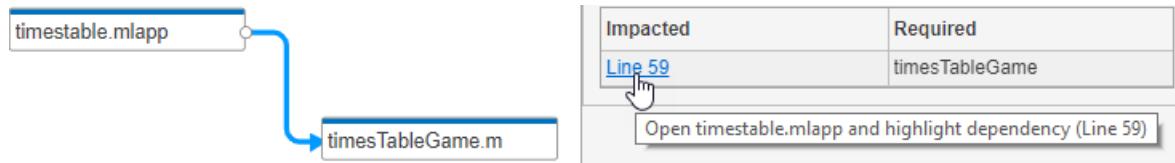
- To zoom in and out, in the **Navigate** section, click **Zoom In** and **Zoom Out**. Alternatively, use the mouse wheel.
- To center and fit the dependency graph to view, in the **Navigate** section, click **Fit to View**. Alternatively, press the **Space** bar.

Investigate Dependency Between Two Files

To see more information about how two files are related, select their dependency arrow. In the **Properties** pane, in the **Details** section, you can see the full paths of the files you are examining, the dependency type (such as function call, inheritance, and property type), and where the dependency is introduced.



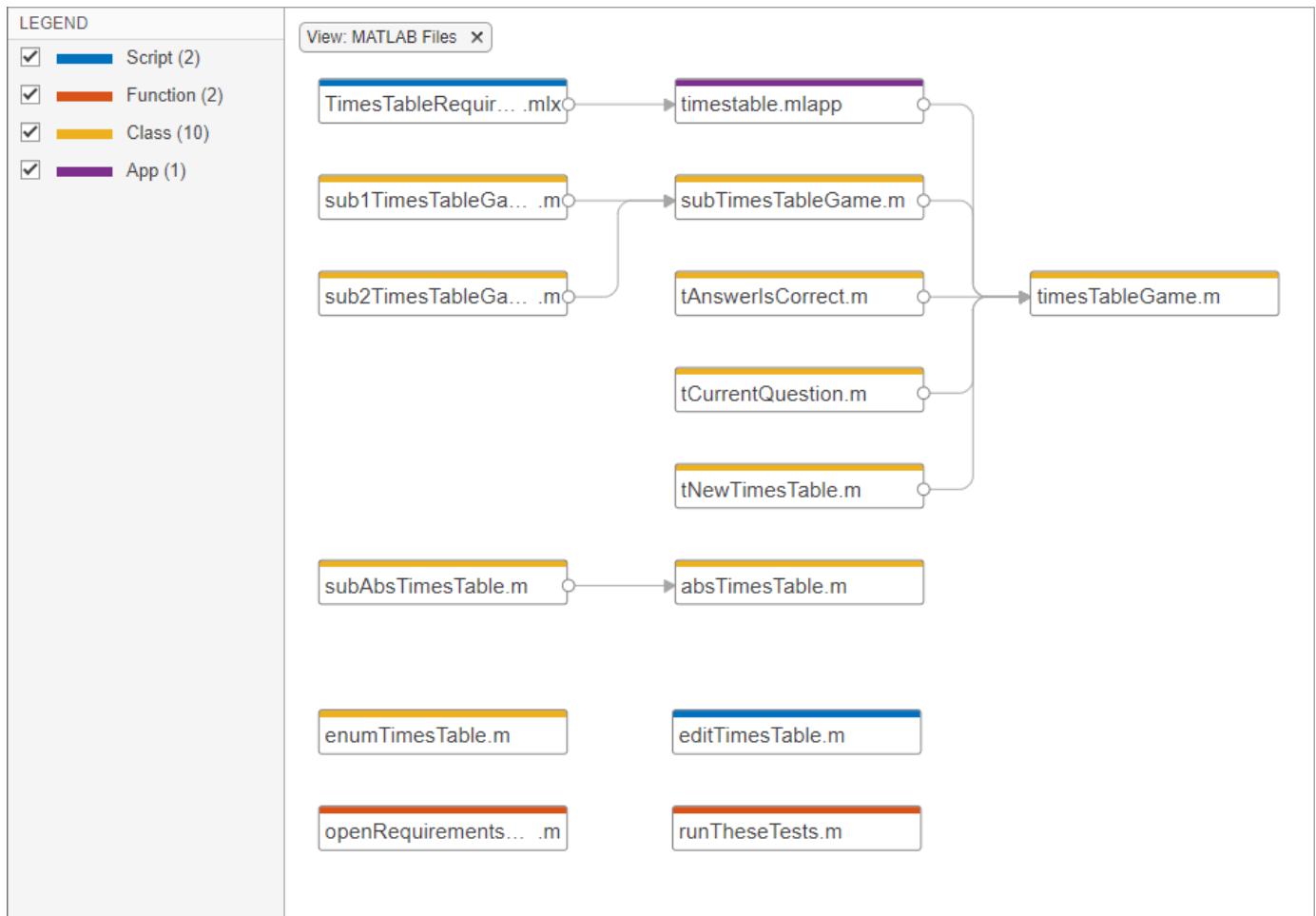
To open the file and highlight where the dependency is introduced, in the **Details** section, click the link under **Impacted**.



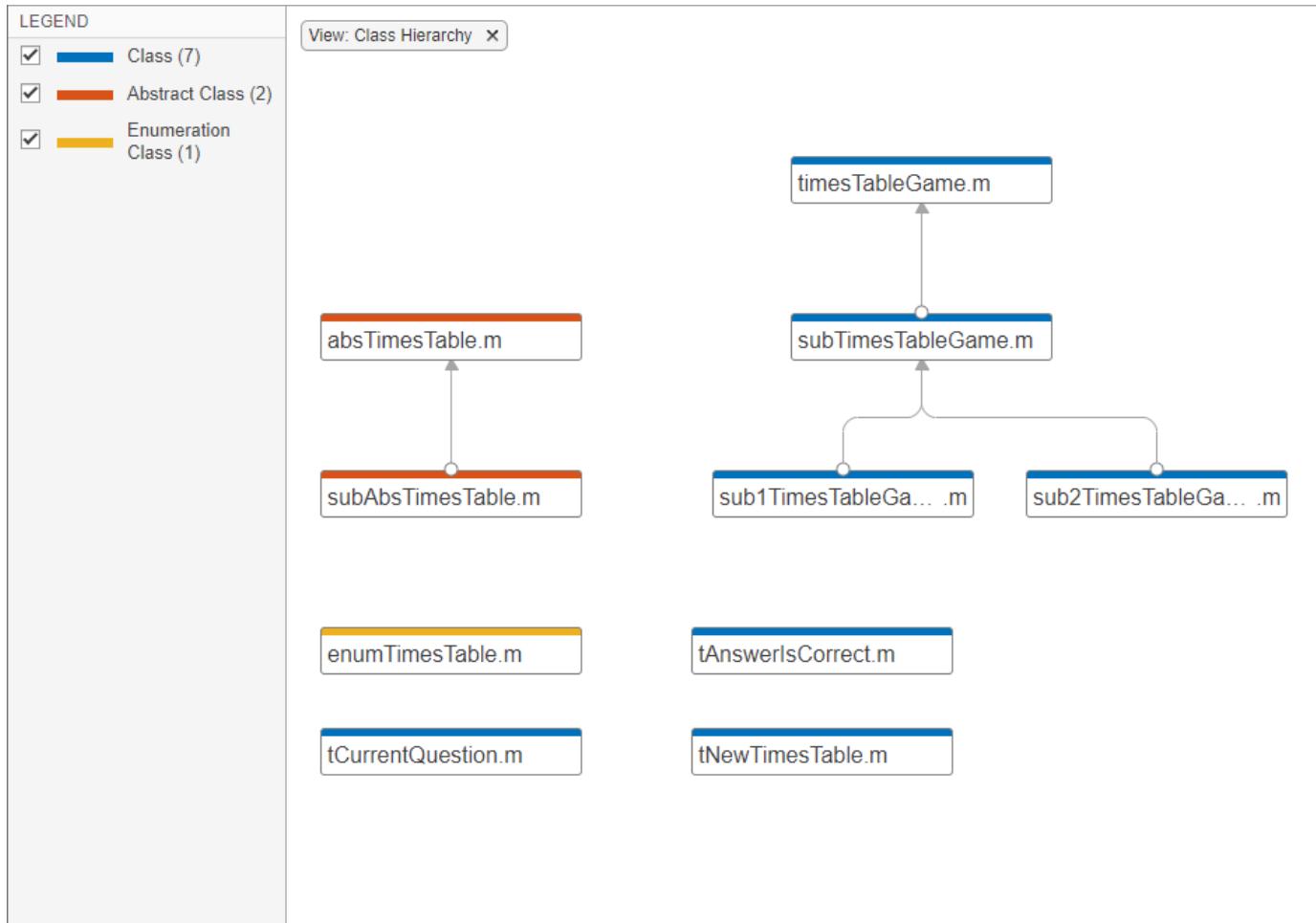
Color Files by Type, Status, or Label

Explore the different views in the **Views** section of the Dependency Analyzer toolbar to explore your project files dependencies.

- The **MATLAB Files** view shows only MATLAB files (such as .m, .mlx, .p, .mlapp, .fig, .mat, and .mex) in the view and colors them by type.

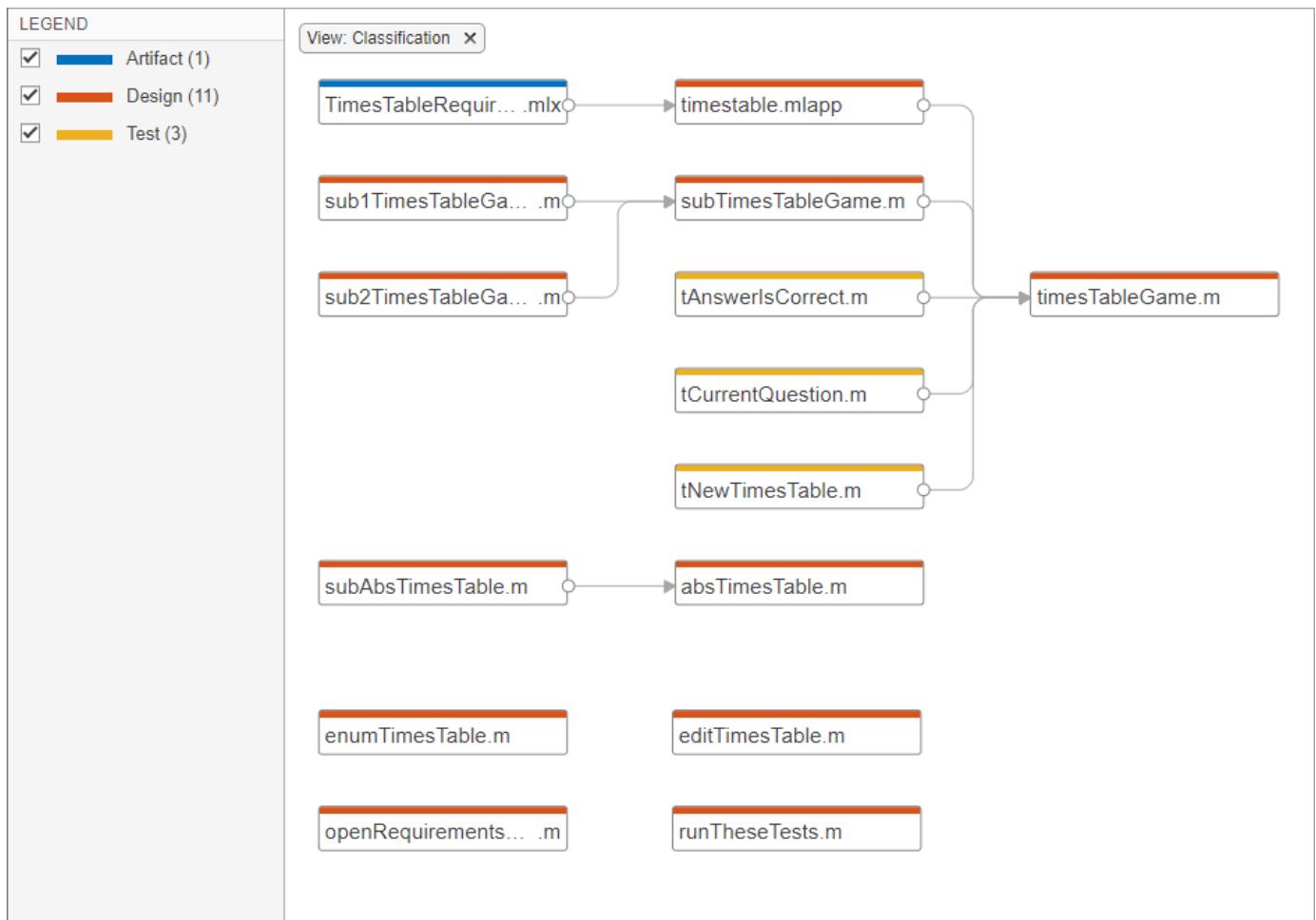


- The **Class Hierarchy** view shows the class inheritance graph and colors the files by type (class, enumeration class, or abstract class). If the class is not on the search path, the Dependency Analyzer cannot determine the class type.



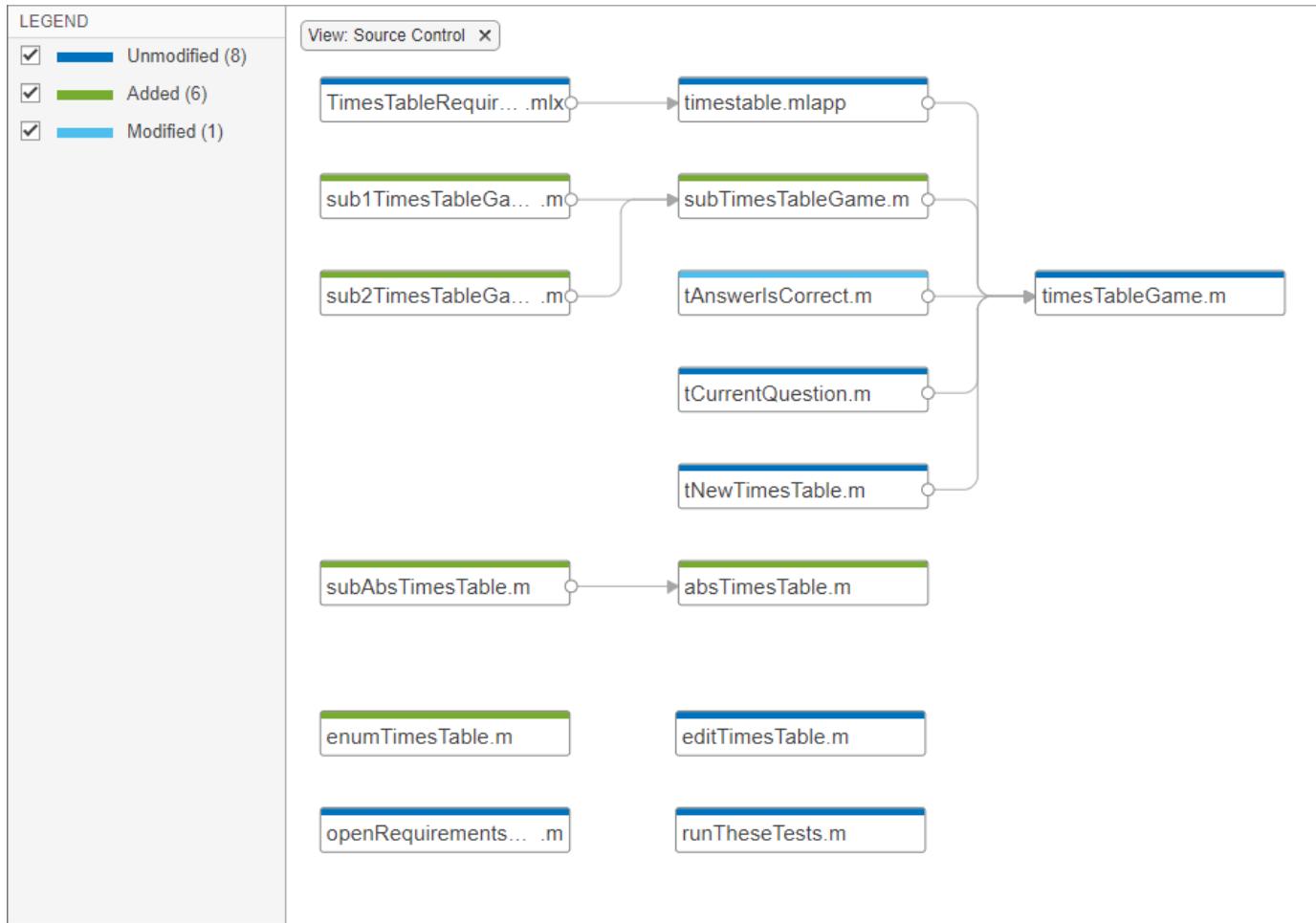
- The **Classification** view shows all files in the graph and colors them by file label (such as test, design, and artifact).

Use the classification view to identify which tests you need to run to validate the changes in your design. For more information, see “Identify Tests to Run” on page 32-41.



- The **Source Control** view shows all files in the graph and colors them by source control status. This view is only enabled if your project is under source control.

Use the source control view to find modified files in your project and to examine the impact of these changes on the rest of the project files. For more information, see “Investigate Impact of Modified Files” on page 32-41.



- **Restore to Default** clears all filters.

This is equivalent to manually removing all of the filters. Filters appear at the top of the graph. For example, if you have the **Source Control** view selected, you can remove it by clicking

View: Source Control X

Apply and Clear Filters

In large projects, when investigating problems or dependencies, use the different filters to show only the files you want to investigate:

- To filter out a subgroup of files from the graph, such as files labeled `test` or modified files, use the check boxes in the **Legend** pane. To remove the legend filter, click the **Legend Filter** **Legend Filter X**.
- To color the files in the graph by type, class, label, or source control status, use the **Views**. To remove the view filter, click **View: *viewName*** at the top of the graph. For example, if you have the **Source Control** view selected, you can remove it by clicking **View: Source Control X**.
- To show only the dependencies of a specific file, select the file and, in the **Impact Analysis** section, click **All Dependencies**. The graph shows the selected file and all its dependencies. To

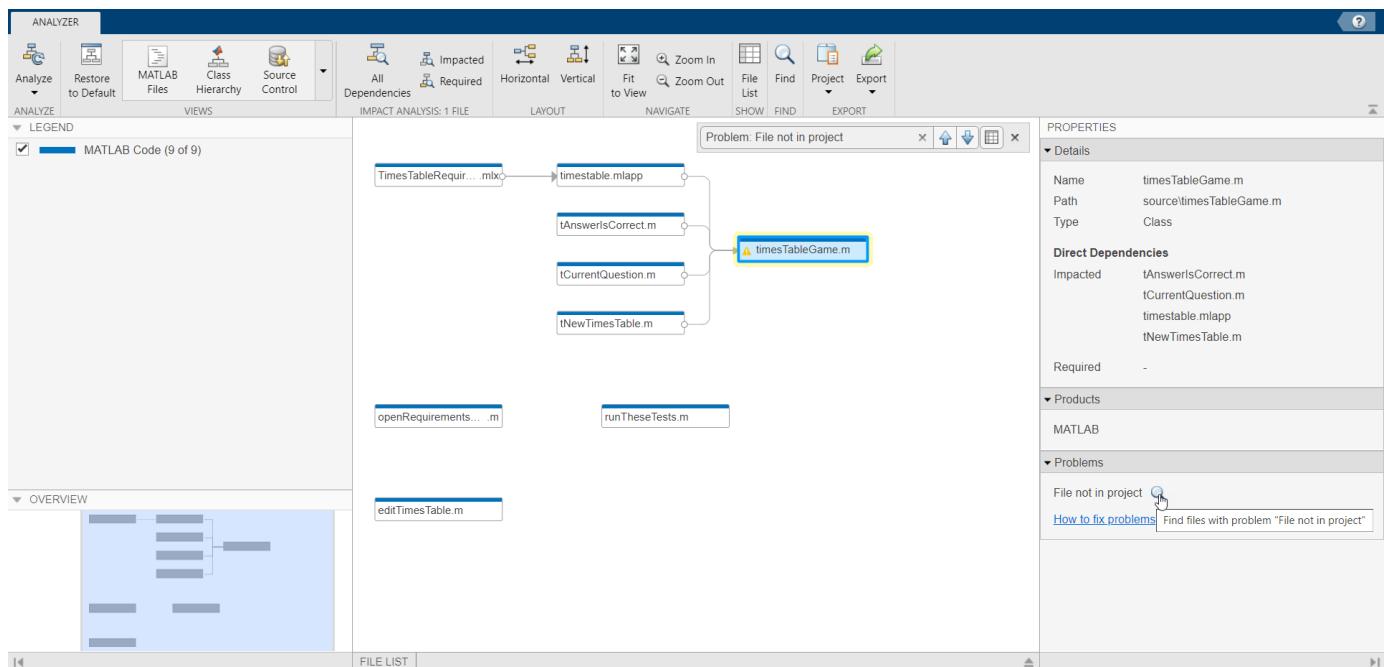
reset the graph to show all project dependencies, remove the filter at the top of the graph. For example, if you filtered by all dependencies of `timestable.mlapp`, to remove the filter click

All dependencies: timestable.mlapp

- To clear all filters and restore the graph to show all analyzed dependencies in the project, click **Restore to Default**. Alternatively, manually remove all filters at the top of the graph.

Investigate and Resolve Problems

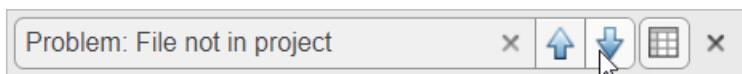
When you run a dependency analysis, the Dependency Analyzer identifies problems, such as missing files, files not in the project, unsaved changes, and out-of-date derived files. You can examine problem files using the dependency graph or the file list. When no file is selected, the **Properties** pane on the right shows the add-ons dependencies and a list of problems for the entire project.



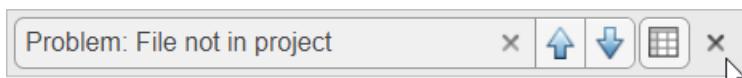
Use the graph to investigate problem files graphically.

- In the **Properties** pane, in the **Problems** section, point to a problem, such as *File not in project*, and click the magnifying glass icon . The graph highlights the files with this specific problem.

To go through these files, use the arrows in the search box (e.g., **Problem: File not in project**).



To undo the highlighting, close the search box.



- 2 To see more information about a specific problem file, select the file in the graph. In the **Properties** pane, in the **Problems** section, you can see details including the path, type, and the problems for this file.

For example, if a file is *File not in project*, right-click the problem file in the graph and select **Add to Project**.

- 3 Investigate the next problem listed in the **Problems** section. Repeat the steps until you resolve all problems. For more details on how to fix problems, see “Resolve Problems” on page 32-38.

To update the graph and the **Problems** list, click **Analyze**.

Tip For large projects, viewing the results in a list can make navigation easier.

For large projects, use the **File List** to investigate your project problem files.

- 1 In the Dependency Analyzer toolbar, click **File List**.
2 In the **Properties** pane, in the **Problems** section, point to a problem, such as *File not in project*, and click the magnifying glass icon .

The **File List** shows only files with the specific problem. Select all the files in the list and use the context menu to **Add to Project**.

- 3 Investigate the next problem listed in the **Problems** section, for example *Missing file*. Repeat the steps until you resolve all problems.

To update the graph and the **Problems** list, click **Analyze**.

Resolve Problems

For each problem file, take actions to resolve the problem. This table lists common problems and describes how to fix them.

Problem Message	Description	Fix
File not in project	The file is not in the project.	Right-click the problem file in the graph and select Add to Project . To remove a file from the problem list without adding it to the project, right-click the file and select Hide Warnings .
Missing file	The file is in the project but does not exist on disk.	Create the file or recover it using source control.

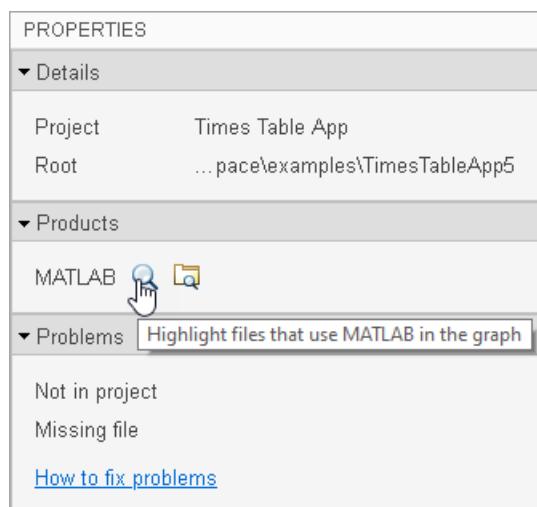
Problem Message	Description	Fix
	The file or variable cannot be found.	<p>If this status is acceptable, right-click the file and select Hide Warnings.</p> <p>Depending on the way you call an object method, the Dependency Analyzer might confuse a method with a function and report a missing dependency. See "Analysis Limitations" (Simulink).</p>
In unreferenced project	The file is within a project that is not referenced by the current project.	Add the project containing the file as a project reference
Outside project root	The file is outside the project root folder.	<p>If this status is acceptable, right-click the file and select Hide Warnings. Otherwise, move it under the project root.</p> <p>If required files are outside your project root, you cannot add these files to your project. This dependency might not indicate a problem if the file is on your path and is a utility or resource that is not part of your project. Use dependency analysis to ensure that you understand the design dependencies.</p>
Unsaved changes	The file has unsaved changes in the MATLAB and Simulink editors.	Save the file.
Derived file out of date	The derived file is older than the source file it was derived from.	<p>Regenerate the derived file. If it is a .p file, you can regenerate it automatically by running the project checks. In MATLAB, on the Project tab, select Run Checks > Check Project and follow the prompts to rebuild the files.</p> <p>If you rename a source file, the project detects the impact to the derived file and prompts you to update it.</p>
Product not installed	The project has a dependency on a missing product.	<p>Install the missing product.</p> <p>Note If you use <code>parfor</code> or <code>spmd</code> but do not have the Parallel Computing Toolbox installed, the corresponding code runs sequentially. The Dependency Analyzer reports a problem in the Problems section.</p>

Find Required Products and Add-Ons

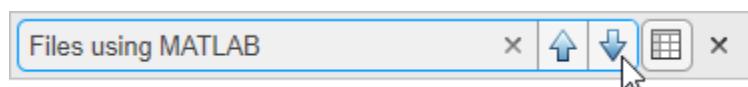
After running a dependency analysis on a project, the graph shows the required add-ons for the whole project or for selected files. You can see which products are required to use the project or find which file is introducing a product dependency.

In the Dependency Analyzer, in the **Properties** pane, the **Product** section displays the required products for the whole project. To view products required by a specific file, select a file by clicking the graph.

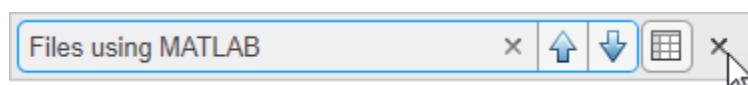
To find which file is introducing a product dependency, point to the product name and click the magnifying glass icon . The graph highlights the files that use the selected product.



To go through these files, use the arrows in the search box (e.g., **Files using "productName"**).



To undo the highlighting, close the search box.



To undo the highlighting, close the search box.

To investigate further, you can list the files that use a product and examine where in these files the dependency is introduced. In the **Products** section, in the **Properties** pane, point to product and click the search folder icon .

If a required product is missing, the products list labels it as missing. The product is also listed in the **Problems** section as **productName not installed**. To resolve a missing product, install the product and rerun the dependency analysis.

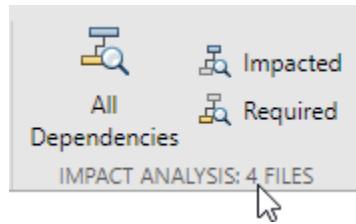
Find File Dependencies

To investigate the dependencies of a file after running a dependency analysis, in the dependency graph, select a file.

- In the **Impact Analysis** section, click **All Dependencies**. The graph shows the selected file and all its dependencies.
- To show only files needed by the selected file to run properly, click **Required**.
- To show only files impacted by a potential change to the selected file, click **Impacted**.

Finding these dependencies can help you identify the impact of a change and identify the tests you need to run to validate your design before committing the changes.

To investigate the dependencies of multiple files, click files while holding the **Shift** key. The **Impact Analysis** section displays how many files are selected.



To reset the graph, click the filter at the top of the graph. For example, if you had filtered by files impacted by `timestable.mlapp`, click `Impacted: timestable.mlapp X`.

Investigate Impact of Modified Files

To examine the impact of the changes you made on the rest of the project files, perform an impact analysis on the modified files in your project.

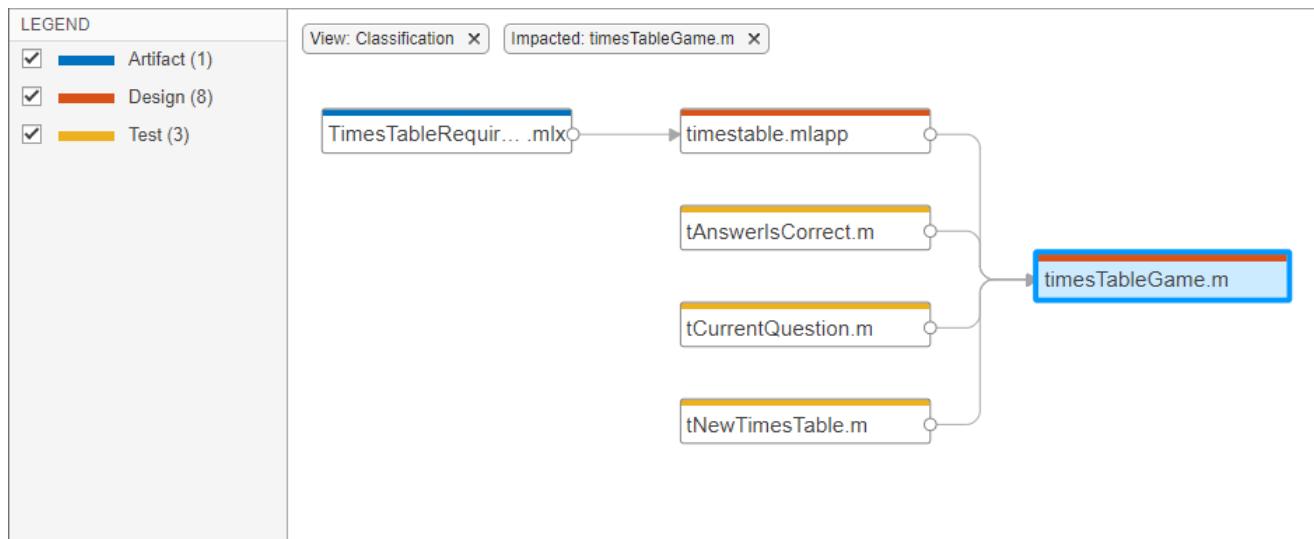
- 1 In the **Views** section, select the **Source Control** view. The graph colors the files by their source control status. The modified files are in light blue.
- 2 Select all the modified files in the graph.

Alternatively, add all modified files to selection by clicking the **Add to selection** icon of an item in the **Legend** pane.

- 3 In the **Impact Analysis** section, click **Impacted**. Alternatively, use the context menu and select **Find Impacted**.

Identify Tests to Run

To identify the tests you need to run to validate your design before committing the changes, use the **Classification** view when you perform an impact analysis on the file you changed.



- 1 In the **Views** section, select the **Classification** view. The graph colors the files by their project label.
- 2 Select the file you changed, for example `timesTableGame.m`.
- 3 In the **Impact Analysis** section, click **Impacted**. Alternatively, use the context menu and select **Find Impacted**.

The example graph shows three tests you need to run to qualify the change made to `timesTableGame.m`.

Export Dependency Analysis Results

To export all the files displayed in the dependency graph, click the graph background to clear the selection on all files. In the Dependency Analyzer toolbar, in the **Export** section, click **Export**. Select from the available options:

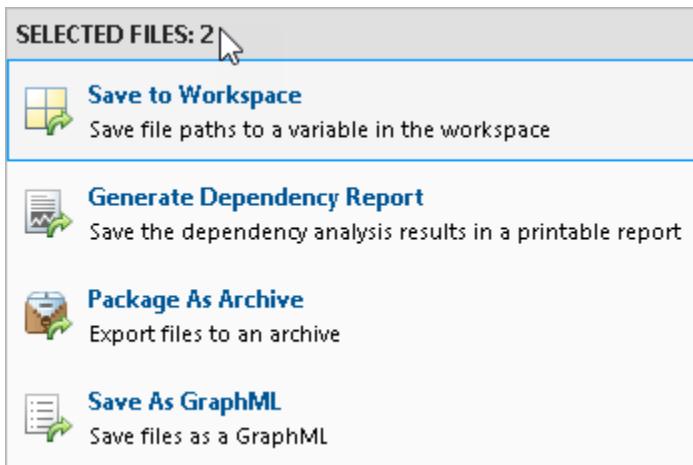
- **Save to Workspace** — Save file paths to a variable in the workspace.
- **Generate Dependency Report** — Save dependency analysis results in a printable report (HTML, Word, or PDF).
- **Package As Archive** — Export files in the graph as an archive.
- **Save As GraphML** — Save dependency analysis results as a GraphML file.

Tip You can compare different analysis results without having to repeat the analysis. To compare previously saved graphs, in MATLAB, in the **Current Folder**, right-click two GraphML files and select **Compare Selected Files/Folders**.

To export a subset of files in the graph, select the files, then click **Export**.

- Use the **Legend** check boxes, the filtered **Views**, or the **Impact Analysis** tools to simplify the graph.
- To select multiple files, press **Shift** and select the files.
- To select all files in the filtered graph, press **Ctrl+A**.

The menu displays how many files are selected. The Dependency Analyzer exports only the selected files.



Note When you use **Package As Archive**, the Dependency Analyzer includes the selected files and all their dependencies in the archive.

Send Files to Project Tools

You can send files to other Project tools using the **Project** menu. The Dependency Analyzer exports only the selected files in the current filtered view.

Select the desired files. In the Dependency Analyzer toolbar, in the **Export** section, click **Project**. Select from the available options:

- **Show in Project** — Switch to the project **Files** view with the files selected.
- **Send to Custom Task** — Run a project custom task on the selected files.

See Also

More About

- “Share Projects” on page 32-23
- “Use Source Control with Projects” on page 32-45

Clone from Git Repository

To create a new project from an existing repository:

- 1 On the **Home** tab, click **New > Project > From Git**. The New Project From Source Control dialog box opens.
- 2 Enter your HTTPS repository path into the **Repository path** field.
- 3 In the **Sandbox** field, select the working folder where you want to put the retrieved files for your new project.
- 4 Click **Retrieve**.

If an authentication dialog box for your repository appears, enter the login information for your Git repository account -- for instance, your GitHub user name and password.

If your repository already contains a project, the project is ready when the tool finishes retrieving files to your selected sandbox folder.

If your sandbox does not yet contain a project, then a dialog box asks whether you want to create a project in the folder. To create a project, specify a project name and click **OK**. The Welcome screen appears to help you set up your new project. For more information about setting up a project, see “Set up Project” on page 32-3.

You can now add, delete, and modify your project files. For details on how to commit and push the modified project files, see “Commit Modified Project Files” on page 32-51.

Tip Alternatively, to prevent frequent login prompts when you interact with your remote repository, you can clone a remote repository using SSH instead of HTTPS. To avoid problems connecting using SSH, set the **HOME** environment variable and use it to store your SSH keys. For more information, see “Use SSH Authentication with MATLAB” on page 33-23.

If you encounter errors like `OutOfMemoryError: Java heap space` when cloning large Git repositories, 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

- “Use SSH Authentication with MATLAB” on page 33-23
- “Commit Modified Project Files” on page 32-51

Use Source Control with Projects

You can use projects to work with files under source control. Source control stores files in a repository, lets you check files out of the repository to work on and back into it to preserve changes, and lets you see the history of the different versions of files that have been checked in. For more information about using source control in MathWorks products, see “About MathWorks Source Control Integration” on page 33-2.

Projects integrate with two source control systems, Git and Subversion (SVN).

To set up a project with source control, use any of these workflows:

- Create a new project from an existing repository.
- Add an existing project to source control.
- Create a new project in a folder already under source control.
- Create a new GitHub repository for a new or an existing project.

Then, when your project is under source control, you can perform operations from within MATLAB such as checking files in and out, running checks, and committing and reverting changes.

Setup Source Control

There are four ways to set up a project with source control.

Create New Project from Existing Repository

Create a new local copy of a project from an existing repository by retrieving files from source control. You can clone a Git repository, or check out files from an SVN repository, or use another source control integration.

To create a new project from an existing repository:

- 1 On the **Home** tab, click **New > Project > From Git** or **New > Project > From SVN**. The New Project From Source Control dialog box opens.
- 2 If you know your repository location, paste it into the **Repository path** field.

Otherwise, to browse for and validate the repository path to retrieve files from, click **Change**.

- a In the dialog box, specify the repository URL by entering or pasting a URL into the field, by selecting from the list of recent repositories, or by clicking the  button.
- b Click **Validate** to check the repository path. If the path is invalid, check the URL against your source control repository browser.
- c If you see an authentication dialog box for your repository, enter login information to continue.
- d If necessary, select a deeper folder in the repository tree. With SVN, you might want to check out from `trunk` or a branch folder under `tags`.
- e When you have finished specifying the URL path you want to retrieve, click **OK**. The dialog box closes and you return to the New Project From Source Control dialog box.

- 3 In the **Sandbox** field, select the working folder where you want to put the retrieved files for your new project. With SVN, use a local folder for best results, because using a network folder is slow.

4 Click Retrieve.

If your repository already contains a project, the project is ready when the tool finishes retrieving files to your selected sandbox folder.

If your sandbox does not yet contain a project, then a dialog box asks whether you want to create a project in the folder. To create a project, specify a project name and click **OK**. The Welcome screen appears to help you set up your new project. For more information about setting up a project, see “Set up Project” on page 32-3.

If you encounter errors like `OutOfMemoryError: Java heap space` when cloning large Git repositories, 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.

Add Existing Project to Source Control

If you have an existing project, you can add it to Git or SVN source control.

To add a project to source control:

- 1** On the **Project** tab, in the **Source Control** section, click **Use Source Control**. The Source Control Information dialog box opens.
- 2** Click the **Add Project to Source Control** button. The Add to Source Control dialog box opens.
- 3** In the **Source control tool** list, select the right tool for your repository. If you choose Git, then skip step 4 and go straight to step 5.
- 4** If you are using an SVN remote repository, click the **Change** button to select an existing repository or create a new one.
 - To specify an existing repository, click the  button to browse for your repository, paste a URL into the field, or use the list to select a recent repository.
 - To create a new repository, click the  button. For more information about creating a new repository, see “Create New Repository” on page 33-5.

Click **Validate** to check the path to the selected repository, and then click **OK**.

- 5** Click **Convert** to finish adding the project to source control.

The project runs integrity checks.

- 6** After the integrity checks run, click **Open Project** to return to your project.

The project displays details of the current source control tool and the repository location.

- 7** If you created a new repository, select the **Files > Modified** view and click **Commit** to commit the first version of your files to the new repository. In the dialog box, enter a comment if you want, and click **Submit**.

If you want to merge branches with Git, you need to follow additional setup steps. For more information, see “Set Up Git Source Control” on page 33-23.

If you want to use a version of SVN other than the built-in version, see “Set Up SVN Source Control” on page 33-14.

Create New Project in Folder Already Under Source Control

If you create a new project from a folder that is already under source control, MATLAB can automatically add the new project to source control. After creating the project, click the **Detect** button. For more information about creating a project from a folder, see “Create Projects” on page 32-2.

Create New GitHub Repository

Creating a GitHub repository adds Git source control to your new or existing project. The GitHub repository you create becomes the project remote repository. To create a GitHub repository, you must have a GitHub account.

To create a blank project and a GitHub remote repository:

- 1 On the **Home** tab, click **New > Project > From Git**.
- 2 Select **New > GitHub Repository**. In the GitHub dialog box, enter your **User name** and **Password**. Fill the **Repository name** and **Description** fields and click **Create**.

MATLAB creates a new public GitHub repository and populates the **Repository path** field with information in the <https://github.com/myusername/mynewrepository> format.

- 3 In the **Sandbox** field, specify the location for your sandbox. The selected folder must be empty. Click **Retrieve** to create the sandbox.

To confirm the project name and creation, click **OK**.

After creating the GitHub repository and sandbox, add your files to the sandbox. See “Mark Files for Addition to Source Control” on page 33-8. Commit the first version of your files to your local repository, then push all modifications to your remote GitHub repository.

Tip If you want to create a remote GitHub repository for an existing project, share your project to GitHub instead.

With your project loaded, on the **Project** tab, select **Share > GitHub**. For detailed instructions, see Publish on GitHub in “Share Projects” on page 32-23.

Perform Source Control Operations

Retrieve Versions and Checkout Project Files

This table shows how to check for modified project files, update revisions, get and manage file locks, and tag project files.

Action	Procedure
Refresh status of project files.	To check for locally modified files, on the Project tab, in the Source Control section, click Refresh . Refreshing queries the local sandbox state and checks for changes made with another tool outside of MATLAB. For more information, see “Update SVN File Status and Revision” on page 33-21 or “Update Git File Status and Revision” on page 33-29.
Check for modifications in project files.	To find out if there is a new version of the project in the repository, in the Files view, right-click the file and select Source Control > Check for Modifications . With SVN, this option contacts the repository to check for external modifications. The project compares the revision numbers of the local file and the repository version. If the revision number in the repository is larger than that in the local sandbox folder, then the project displays (<i>not latest</i>) next to the revision number of the local file.
Update all project files.	Using SVN, to get the latest changes of all project files, go to the Project tab, and in the Source Control section, click Update . The project displays a dialog box listing all the files that have changed on disk. You can control this behavior using the project preference Show changes on source control update . For more information, see “Update SVN File Status and Revision” on page 33-21. Using Git, to get the latest changes for all project files from a source control repository and merge them into your current branch, go to the Project tab, and in the Source Control section, click Pull . To get changes and merge manually, on the Project tab, in the Source Control section, click Fetch . This updates all of the origin branches in the local repository. When you click Fetch , your sandbox files are not changed. To see the changes from others, merge in the origin changes to your local branches. For more information, see “Pull, Push and Fetch Files with Git” on page 33-34.
Update revision for selected project files.	To update a selected set of files, in the Files view, right-click the files, and select the Source Control > Update command for the source control system you are using. For example, if you are using SVN, select Source Control > Update from SVN to get fresh local copies of the selected files from the repository.
Clear SVN login information.	To clear any stored login credentials, on the Project tab, in the Source Control section, click Clear Login .

Action	Procedure
Get SVN file locks.	To get SVN file locks, in the Files view, select the files that you want to check out. Right-click the selected files and select Source Control > Get File Lock . A lock symbol appears in the SVN source control column. Other users do not 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, on the Project tab, click Locks . Get File Lock is only for SVN. Git does not have locks.
Manage SVN repository locks.	To manage global SVN locks for a repository, on the Project tab, in the Source Control section, click Locks . For more information, see “Get SVN File Locks” on page 33-22.
Tag versions of project files.	To identify specific revisions of all project files, on the Project tab, in the Source Control section, click Tag . Specify the tag text and click OK . The tag is added to every project file. Errors appear if you do not have a tags folder in your repository. For more information, see “Set Up SVN Source Control” on page 33-14.

Review Changes in Project Files

You can review changes in project files using the **Files > Modified** view. This table shows how to view the list of modified project files, review the history of a file, and compare two files' revisions.

Action	Procedure
View modified project files.	In the Files view, select the Modified (number of files) tab. The Files > Modified view is visible only if you are using source control with your project. Tip Use the List layout to view files without needing to expand folders.
	You can identify modified or conflicted folder contents using the source control summary status. In the Files view, folders display rolled-up source control status. This makes it easier to locate changes in files, particularly conflicted files. You can hover over the source control status (for instance, the Git or SVN column) for a folder to view a tooltip displaying how many files inside are modified, conflicted, added, and deleted.
Update modified files list.	To update the modified files list, on the Project tab, in the Source Control section, click Refresh .

Action	Procedure
View revision history.	<p>In the Files view, right-click a file and select Source Control > Show Revisions.</p> <p>To browse and compare files within committed SVN change sets, on the Project tab, in the Source Control section, select Show Log. In the File Revisions dialog box, select a revision to view a list of modified files. Right-click files in the lower list to view changes or save revisions.</p>
Compare revisions.	<p>In the Files view, right-click a file and select Compare > Compare to Ancestor to run a comparison against the local repository (Git) or against the last checked-out version in the sandbox (SVN). The Comparison Tool displays a report.</p> <p>To compare other revisions of a file, select Compare > Compare to Revision.</p> <p>To view a comparison report, select the revisions you want to compare and click Compare Selected. Or, select a revision and click Compare to Local. For more information, see “Compare Files and Folders and Merge Files”.</p>

Project Definition Files

The files in the **resources/project** folder are project definition files generated when you first create or make changes to your project. The project definition files enable you to add project metadata to files without checking them out. Some examples of metadata you can change this way are shortcuts, labels, and project descriptions. Project definition files also specify the files added to your project. These files are not part of the project.

Any changes you make to your project generate changes in the **resources/project** folder. These files store the definition of your project in XML files whose format is subject to change.

You do not need to view project definition files directly, except when the source control tool requires a merge. The files are shown so that you know about all the files being committed to the source control system.

Starting in R2020b, the default project definition file type is **Use multiple project files (fixed-path length)**. To change the project definition file management from the type selected when the project was created, use `matlab.project.convertDefinitionFiles`.

`matlab.project.convertDefinitionFiles` preserves the source control history of your project.

Warning To avoid merge issues, do not convert the definition file type more than once for a project.

For releases before R2020b, if you want to change the project definition file management from the type selected when the project was created:

- 1 On the **Home** tab, in the **Environment** section, click **Preferences**. Select **MATLAB > Project** and in the **New Projects** section, select one of the options under **Project definition files**:
 - **Use multiple project files** - Helps to avoid merging issues on shared projects

- **Use multiple project files (fixed-path length)** - Is better if you need to work with long paths
 - **Use a single project file (not recommended for source control)** - Is faster but is likely to cause merge issues when two users submit changes in the same project to a source control tool
- 2** Create a project archive file (.mlproj). For more information, see “Share Projects” on page 32-23 or **export**.
- 3** Create a new project from the archived project. For more information, see “Create Projects” on page 32-2.

To stop managing your folder with a project and delete the resources/project folder, see `matlab.project.deleteProject`.

Run Project Checks

To run checks for a project, go to the **Project** tab and click **Run Checks > Check Project**. The project checks for problems with project integrity such as missing files, unsaved files, or files not under source control. A dialog box reports the results. You can click for details and follow prompts to fix problems.

If you want to check for required files, click **Dependency Analyzer** to analyze the dependencies of the modified files. Use the dependency tools to analyze the structure of your project.

For details on problems the checks can fix, see “Work with Derived Files in Projects” on page 32-54, and “Analyze Project Dependencies” on page 32-30.

Commit Modified Project Files

After reviewing changes and running project checks, you are ready to commit your modified project files to source control. This table shows how to commit modified project files.

Action	Procedure
Commit all modified files to source control.	<p>In the Files view, select the Modified (number of files) tab. On the Project tab, in the Source Control section, click Commit. Enter comments in the dialog box, and click Submit. If you are using Git source control, this commits to your local repository. If you are using SVN source control, this commits changes to your repository.</p> <p>A message appears if you cannot commit because the repository has moved ahead. Before you can commit the file, you must update its revision up to the current HEAD revision. If you are using Git source control, click Pull. If you are using SVN source control, click Update. Resolve any conflicts before you commit.</p>
Commit selected files to source control.	<p>In the Files view, select the files, right-click, and select Source Control > Commit.</p> <p>If you commit individual files, you risk not committing the related project definition files that keep track of your files. To avoid this, commit all modified files.</p>

Action	Procedure
Push project files with Git.	To send local commits to the remote repository, on the Project tab, in the Source Control section, click Push . A message appears if you cannot push your changes directly because the repository has moved on. Click Fetch to fetch all changes from the remote repository. Merge branches and resolve conflicts, and then you can push your changes. For more information, see “Pull, Push and Fetch Files with Git” on page 33-34.
Push empty project folders with Git.	You cannot add empty folders to Git source control, so you cannot click Push and then clone an empty folder. You can create an empty folder in a project, but if you push changes and then sync a new sandbox, the empty folder does not appear in the new sandbox. Instead, run Check Project , which creates the empty folder for you. Alternatively, to push empty folders to the repository for other users to sync, create a <code>gitignore</code> file in the folder and then push your changes.
Create Git stashes.	Git stashes store uncommitted changes for later use. To create a stash, on the Project tab, in the Source Control section, click Stashes . The Stashes dialog box opens. Click New Stash to create a stash containing your currently modified files. For more information, see “Use Git Stashes” on page 33-35.
Create a branch with Git.	To create a branch, on the Project tab, in the Source Control section, click Branches . The Branches dialog box appears, where you can view, switch, create, and merge branches. 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 values to create a branch from the head of the current branch. Enter a name in the Branch name text box and click Create .
Switch, compare, save, and merge branches with Git.	To switch, compare, save, and merge branches, on the Project tab, in the Source Control section, click Branches . The Branches dialog box appears, where you can view, switch, create, and merge branches. For more information, see “Branch and Merge with Git” on page 33-30.

Action	Procedure
Resolve conflicts.	<p>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. Extract conflict markers if necessary, compare the differences causing the conflict, and resolve the conflict.</p> <p>Look for conflicted files in the Files > Modified view. Identify conflicted folder contents using the source control summary status. Folders display the rolled-up source control status. This makes it easier to locate changes in files, particularly conflicted files. You can hover over the source control status for a folder to view a tooltip displaying how many files inside are modified, conflicted, added, and deleted.</p> <p>Tip Use the List layout to view files without needing to expand folders.</p> <p>Check the source control status column (Git or SVN) for files with a red warning symbol, which indicates a conflict. Right-click the conflicted file and select View Conflicts to compare versions. A comparison report opens showing the differences between the conflicted files.</p> <p>When you have resolved the changes and want to commit the version in your sandbox, right-click the file and select Source Control > Mark Conflict Resolved.</p> <p>For Git, the Branch status in the Git pane changes from MERGING to SAFE.</p> <p>Select the Files > Modified view to check changes.</p>

Revert Changes

This table shows how to revert changes in project files. For more information about reverting changes, see “Revert Changes in Source Control” on page 33-13.

Action	Procedure
Revert local changes.	<p>To release locks and revert to the version in the last sandbox update (that is, the last version you synchronized or retrieved from the repository), in the Files view, right-click the files to revert and select Source Control > Discard Local Changes and Release Locks.</p> <p>To discard local changes when using Git, right-click a file and select Source Control > Revert Local Changes. To remove all local changes, click Branches in the Git pane and click Revert to Head.</p>

Action	Procedure
Revert a file to a specified revision	<p>To revert a file to a specified revision, right-click a file and select Source Control > Revert using SVN or Source Control > Revert using Git.</p> <p>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, log message. Click Revert.</p> <p>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.</p>
Revert a project to a specified revision.	<p>To revert a project to a specified revision, on the Project tab, in the Source Control section, click Revert Project. In the Revert Files dialog box, choose a revision to revert to.</p> <p>Each revision in the list is a change set of modified files. Select a revision to view information about the change such as the author, date, and the log message.</p> <p>Once you have chosen a revision and are satisfied that the information is correct, click Revert.</p>

Work with Derived Files in Projects

In general, it is a best practice to omit derived and temporary files from your project or exclude them from source control. On the **Project** tab, select **Run Checks > Check Project** to check for derived or temporary files. If you add the `slprj` folder to a project, the project checks advise you to remove this from the project and offer to make the fix.

It is also a best practice to exclude derived files such as `.mex*`, the contents of the `slprj` folder, `sccprj` folder, or other code generation folders from source control, because they can cause problems. For example:

- With a source control system that can do file locking, you can encounter conflicts. If `slprj` is under source control and you generate code, most of the files under `slprj` change and become locked. Other users cannot generate code because of file permission errors. The `slprj` folder is also used for simulation via code generation, so locking these files can have an impact on a team. The same problems arise with binaries, such as `.mex*`.
- Deleting `slprj` is often required. However, deleting `slprj` causes problems such as “not a working copy” errors if the folder is under some source control tools (for example, SVN).
- If you want to check in the generated code as an artifact of the process, it is common to copy some of the files out of the `slprj` cache folder and into a separate location that is part of the project. That way, you can delete the temporary cache folder when you need to. Use the `packNGo` function to list the generated code files, and use the project API to add them to the project with appropriate metadata.
- The `slprj` folder can contain many small files. This can affect performance with some source control tools when each of those files is checked to see if it is up-to-date.

Find Project Files With Unsaved Changes

You can check your project for files with unsaved changes. On the **Project** tab, in the **Tools** section, select **Run Checks > Show Unsaved Changes**.

In the Unsaved Changes dialog box, you can see the project files with unsaved changes. Project only detects unsaved changes edited in the MATLAB and Simulink editors. Manually examine changes edited in other tools. If you have referenced projects, files are grouped by project. You can save or discard all detected changes.

Manage Open Files When Closing a Project

When you close a project, if there are files with unsaved changes, a message prompts you to save or discard changes. You can see all files with unsaved changes, grouped by project if you have referenced projects. To avoid losing work, you can save or discard changes by file, by project, or globally.

To control this behavior, on the **Home** tab, in the **Environment** section, click  **Preferences**. Go to **MATLAB > Project** and in the **Project Shutdown** section, select or clear the check box labeled **Check for open project models and close them, unless they are dirty**.

See Also

More About

- “About MathWorks Source Control Integration” on page 33-2
- “Set Up Git Source Control” on page 33-23
- “Set Up SVN Source Control” on page 33-14

Create and Edit Projects Programmatically

This example shows how to use the Project API to create a new project and automate project tasks for manipulating files. It covers how to create a project from the command line, add files and folders, set up the project path, define project shortcuts, and create a reference to the new project in another project. It also shows how to programmatically work with modified files, dependencies, shortcuts, and labels.

Set up the Example Files

Create a working copy of the Times Table App example project files and open the project. MATLAB® copies the files to an examples folder so that you can edit them. The project puts the files under Git™ source control. Use `currentProject` to create a project object from the currently loaded project.

```
matlab.project.example.timesTable  
mainProject = currentProject;
```

Examine Project Files

Examine the files in the project.

```
files = mainProject.Files  
  
files=1x14 object  
 1x14 ProjectFile array with properties:  
  
  Path  
  Labels  
  Revision  
  SourceControlStatus
```

Use indexing to access files in this list. For example, get file number 10. Each file has properties describing its path and attached labels.

```
mainProject.Files(10)  
  
ans =  
  ProjectFile with properties:  
  
    Path: "C:\workSpace\examples\TimesTableApp\tests\tNewTimesTable.m"  
    Labels: [1x1 matlab.project.Label]  
    Revision: "51316c67b968d45a17e127003a25143577ec011a"  
    SourceControlStatus: Unmodified
```

Get Git latest revision of the tenth file.

```
mainProject.Files(10).Revision  
  
ans =  
"51316c67b968d45a17e127003a25143577ec011a"
```

Examine the labels of the tenth file.

```
mainProject.Files(10).Labels  
  
ans =  
  Label with properties:
```

```

    File: "C:\workSpace\examples\TimesTableApp1\tests\tNewTimesTable.m"
    DataType: 'none'
    Data: []
    Name: "Test"
    CategoryName: "Classification"

```

Get a particular file by name.

```

myfile = findFile(mainProject,"source/timestable.mlapp")

myfile =
  ProjectFile with properties:

    Path: "C:\workSpace\examples\TimesTableApp1\source\timestable.mlapp"
    Labels: [1x1 matlab.project.Label]
    Revision: "51316c67b968d45a17e127003a25143577ec011a"
    SourceControlStatus: Unmodified

```

Create New Project

Create the Times Table Game project. This project will store the game logic behind the Times Table App. The Times Table Game project will be used by the Times Table App project through a project reference.

Create the project and set the project name.

```

timesTableGameFolder = fullfile(mainProject.RootFolder,"refs","TimesTableGame");
timesTableGame = matlab.project.createProject(timesTableGameFolder);
timesTableGame.Name = "Times Table Game";

```

Move the Times Table App game logic from the main project folder to the new project folder, and add it to the Times Table Game project. Then, remove the file from the Times Table App project.

```

movefile("../..\source\timesTableGame.m");
addFile(timesTableGame,"timesTableGame.m");

reload(mainProject);
removeFile(mainProject,"source\timesTableGame.m");

```

Add the Times Table Game project root folder to the Times Table Game project path. This makes the `timesTableGame.m` file available when the Times Table App project or any project that references the Times Table App project is loaded.

```

reload(timesTableGame);
addPath(timesTableGame,timesTableGame.RootFolder);

```

Add a Project Reference

Add the new Times Table Game project to the Times Table App project as a project reference. This allows the Time Table App project to view, edit, and run files in the Times Table Game project.

```

reload(mainProject);
addReference(mainProject,timesTableGame);

```

Get Modified Files

Get all the modified files in the Times Table App project. Compare this list with the **Files > Modified** view in the project. You can see the files for the new Times Table Game project, as well as the removed and modified files in the Times Table App project.

```
modifiedfiles = listModifiedFiles(mainProject)

modifiedfiles=1x8 object
  1x8 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Get the second modified file in the list. Observe that the `SourceControlStatus` property is `Added`. The `listModifiedFiles` function returns any files that are added, modified, conflicted, deleted, and so on.

```
modifiedfiles(2)

ans =
  ProjectFile with properties:

    Path: "C:\workSpace\examples\TimesTableApp1\refs\TimesTableGame\resources\pro...
    Revision: ""
    SourceControlStatus: Added
```

Refresh the source control status before querying individual files. You do not need to do this before calling `listModifiedFiles`.

```
refreshSourceControl(mainProject)
```

Get all the project files that are `Unmodified`. Use the `ismember` function to get an array of logicals stating which files in the Times Table App project are unmodified. Use the array to get the list of unmodified files.

```
unmodifiedStatus = ismember([mainProject.Files.SourceControlStatus],matlab.sourcecontrol.Status.Unmodified)
mainProject.Files(unmodifiedStatus)

ans=1x9 object
  1x9 ProjectFile array with properties:

    Path
    Labels
    Revision
    SourceControlStatus
```

Get File Dependencies

Run a dependency analysis to update the known dependencies between project files.

```
updateDependencies(mainProject)
```

Get the list of dependencies in the Times Table App project. The `Dependencies` property contains the graph of dependencies between project files, stored as a MATLAB `digraph` object.

```
g = mainProject.Dependencies

g =
    digraph with properties:

    Edges: [5×1 table]
    Nodes: [9×1 table]
```

Get the files required by the `timestable.mlapp` file.

```
requiredFiles = bfsearch(g, which('source/timestable.mlapp'))

requiredFiles = 2×1 cell
    {'C:\workSpace\examples\TimesTableApp1\source\timestable.mlapp' }
    {'C:\workSpace\examples\TimesTableApp1\refs\TimesTableGame\timesTableGame.m' }
```

Get the top-level files of all types in the graph. The `indegree` function finds all the files that are not depended on by any other file.

```
top = g.Nodes.Name(indegree(g)==0)

top = 7×1 cell
    {'C:\workSpace\examples\TimesTableApp1\requirements\TimesTableRequirements mlx' }
    {'C:\workSpace\examples\TimesTableApp1\tests\tAnswerIsCorrect.m' }
    {'C:\workSpace\examples\TimesTableApp1\tests\tCurrentQuestion.m' }
    {'C:\workSpace\examples\TimesTableApp1\tests\tNewTimesTable.m' }
    {'C:\workSpace\examples\TimesTableApp1\utilities\editTimesTable.m' }
    {'C:\workSpace\examples\TimesTableApp1\utilities\openRequirementsDocument.m' }
    {'C:\workSpace\examples\TimesTableApp1\utilities\runTheseTests.m' }
```

Get the top-level files that have dependencies. The `indegree` function finds all the files that are not depended on by any other file, and the `outdegree` function finds all the files that have dependencies.

```
top = g.Nodes.Name(indegree(g)==0 & outdegree(g)>0)

top = 4×1 cell
    {'C:\workSpace\examples\TimesTableApp1\requirements\TimesTableRequirements mlx' }
    {'C:\workSpace\examples\TimesTableApp1\tests\tAnswerIsCorrect.m' }
    {'C:\workSpace\examples\TimesTableApp1\tests\tCurrentQuestion.m' }
    {'C:\workSpace\examples\TimesTableApp1\tests\tNewTimesTable.m' }
```

Find impacted (or "upstream") files by creating a transposed graph. Use the `flipedge` function to reverse the direction of the edges in the graph.

```
transposed = flipedge(g)

transposed =
    digraph with properties:

    Edges: [5×1 table]
    Nodes: [9×1 table]
```

```
impacted = bfsearch(transposed,which('source/timetable.mlapp'))  
impacted = 2x1 cell  
{'C:\workSpace\examples\TimesTableApp1\source\timetable.mlapp'}  
{'C:\workSpace\examples\TimesTableApp1\requirements\TimesTableRequirements mlx'}
```

Get information on the project files, such as the number of dependencies and orphans.

```
averageNumDependencies = mean(outdegree(g));  
number0fOrphans = sum(indegree(g)+outdegree(g)==0);
```

Change the sort order of the dependency graph to show project changes from the bottom up.

```
ordered = g.Nodes.Name(flip(toposort(g)));
```

Query Shortcuts

You can use shortcuts to save frequent tasks and frequently accessed files, or to automate startup and shutdown tasks.

Get the Times Table App project shortcuts.

```
shortcuts = mainProject.Shortcuts  
  
shortcuts=1x4 object  
1x4 Shortcut array with properties:  
  
Name  
Group  
File
```

Examine a shortcut in the list.

```
shortcuts(2)  
  
ans =  
Shortcut with properties:  
  
Name: "Edit Times Table App"  
Group: "Launch Points"  
File: "C:\workSpace\examples\TimesTableApp1\utilities\editTimesTable.m"
```

Get the file path of a shortcut.

```
shortcuts(3).File  
  
ans =  
"C:\workSpace\examples\TimesTableApp1\utilities\openRequirementsDocument.m"
```

Examine all the files in the shortcuts list.

```
{shortcuts.File}'  
  
ans=4x1 cell array  
{{"C:\workSpace\examples\TimesTableApp1\source\timetable.mlapp"}  
{{"C:\workSpace\examples\TimesTableApp1\utilities\editTimesTable.m"}  
{{"C:\workSpace\examples\TimesTableApp1\utilities\openRequirementsDocument.m"}}
```

```
{[ "C:\workSpace\examples\TimesTableApp1\utilities\runTheseTests.m" ]}
```

Label files

Create a new category of labels of type `char`. In the Times Table App project, the new **Engineers** category appears in the **Labels** pane.

```
createCategory(mainProject, 'Engineers', 'char')

ans =
    Category with properties:

        Name: "Engineers"
        SingleValued: 0
        DataType: "char"
        LabelDefinitions: [1x0 matlab.project.LabelDefinition]
```

Define a new label in the new category.

```
category = findCategory(mainProject, 'Engineers');
createLabel(category, 'Bob');
```

Get the label definition object for the new label.

```
ld = findLabel(category, 'Bob')

ld =
    LabelDefinition with properties:

        Name: "Bob"
        CategoryName: "Engineers"
```

Attach a label to a project file. If you select the file in the Times Table App project, you can see this label in the **Label Editor** pane.

```
myfile = findFile(mainProject, "source/timestable.mlapp");
addLabel(myfile, 'Engineers', 'Bob');
```

Get a particular label and attach text data to it.

```
label = findLabel(myfile, 'Engineers', 'Bob');
label.Data = 'Email: Bob.Smith@company.com'

label =
    Label with properties:

        File: "C:\workSpace\examples\TimesTableApp1\source\timestable.mlapp"
        DataType: 'char'
        Data: 'Email: Bob.Smith@company.com'
        Name: "Bob"
        CategoryName: "Engineers"
```

Retrieve the label data and store it in a variable.

```
mydata = label.Data
```

```
mydata =  
'Email: Bob.Smith@company.com'
```

Create a new label category with data type **double**, the type MATLAB commonly uses for numeric data.

```
createCategory(mainProject, 'Assessors', 'double');  
category = findCategory(mainProject, 'Assessors');  
createLabel(category, 'Sam');
```

Attach the new label to a specified file and assign data value 2 to the label.

```
myfile = mainProject.Files(10);  
addLabel(myfile, 'Assessors', 'Sam', 2)  
  
ans =  
    Label with properties:  
  
        File: "C:\workSpace\examples\TimesTableApp\utilities"  
        DataType: 'double'  
        Data: 2  
        Name: "Sam"  
        CategoryName: "Assessors"
```

Close Project

Close the project to run shutdown scripts and check for unsaved files.

```
close(mainProject)
```

See Also

[currentProject](#)

More About

- “Create Projects” on page 32-2
- “Manage Project Files” on page 32-10
- “Analyze Project Dependencies” on page 32-30

Explore an Example Project

This example uses the Times Table App example project to explore how project tools can help you organize your work.

Using the Times Table App example, we will explore how to:

- 1 Set up and browse some example project files under source control.
- 2 Examine project shortcuts to access frequently used files and tasks.
- 3 Analyze dependencies in the project and locate required files that are not yet in the project.
- 4 Modify some project files, find and review modified files, compare them to an earlier version, and commit modified files to source control.
- 5 Explore views of project files only, modified files, and all files under the project root folder.

Setup the Example Files

Create a working copy of the Times Table App example project files and open the project. MATLAB® copies the files to an examples folder so that you can edit them. The project puts the files under Git™ source control.

```
matlab.project.example.timesTable
```

View, Search, and Sort Project Files

You can view, search, and sort project files by using the **Files** view.

To view the files in the project, in the **Files** view, click **Project (number of files)**. When the view is selected, only the files in your project are shown.

To see all the files in your project folder, click **All**. This view shows all the files that are under the project root, not just the files that are in the project. As a result, this view is useful for adding files to the project.

To view files as a list instead of a tree, in the **Layout** field at the top right of the **Files** view, select **List**.

There are several ways to find files and folders in projects:

- To search for particular files or file types by name, in any file view, type in the search box or click the **Filter** button. For example, in the search field, enter the text `timestable`. The project returns all files and folders that contain the word `timestable`. Click the  to clear the search.
- To search the content of files, go to the **Project** tab and click the **Search** button. Enter a value in the search field and click **Enter**. For example, enter the word `tests`. The project displays all files and folders that contain the word `tests`. Click the  to clear the search.
- To change how files are grouped or sorted, and to customize the columns, click the actions  button and select from the available options.

Open and Run Frequently Used Files

You can use shortcuts to make files easier to find in a large project. View and run shortcuts on the **Project Shortcuts** tab. You can organize the shortcuts into groups.

The Times Table App project contains several shortcuts, including a shortcut to open the project requirements, and another to run all the tests in the project. The shortcuts make these tasks easier for users of the project.

To perform an action, on the **Project Shortcuts** tab, click the associated shortcut. For example, to open project requirements, click **Documentation > Requirements**. To run tests, click **Test > Run All Tests**.

To create a new shortcut, select the **Files** view, right-click a file, and select **Create Shortcut**.

Add Folder to Project

Create a new folder and add it to the project path. Adding a project folder to the project path ensures that all users of the project can access the files within it.

- 1 Select the **Files** view.
- 2 Right-click in white space and then select **New > Folder**. Enter a name for the folder. The folder is automatically added to the project.
- 3 Right-click the new folder and select **Project Path > Add to the Project Path (Including Subfolders)**.

Review Changes in Modified Files

Open files, make changes, and review the changes.

- 1 Select the **Files** view. View folders using the tree layout, and then expand the **utilities** folder.
- 2 Right-click **source/timesTableGame.m** and select **Open**.
- 3 Make a change in the Editor, such as adding a comment, and save the file.
- 4 In the **Files** view, select the **Modified (number of files)** tab. After editing the file, you see **Modified (2)**. The file you changed appears in the list.
- 5 To review changes, right-click **source/timesTableGame.m** in the **Modified** files view and select **Compare > Compare to Ancestor**. The MATLAB Comparison Tool opens a report comparing the modified version of the file in your sandbox to its ancestor stored in version control. The comparison report type can differ depending on the file you select. If you select a Simulink® model to compare, this command runs a Simulink model comparison.

* Note - When you open the Times Table App example project, the project shows a modified file in the **resources** folder. This is a side effect of opening the example project. When editing files in your own projects, only changes that affect file metadata, such as adding a label to a file, create modified files in the **resources** folder.

Analyze Dependencies

To check that all required files are in the project, run a file dependency analysis on the modified files.

- 1 On the **Project** tab, click **Dependency Analyzer**.
- 2 The dependency graph displays the structure of all analyzed dependencies in the project. The right pane lists required add-ons and any problem files. Observe that there are no problems files listed.

Now, remove one of the required files. Select the project **Files** view, right click the **source/timesTableGame.m** file, and select **Remove from Project**. Click **Remove** in the Remove from Project dialog box.

The Dependency Analyzer automatically updates the graph and the **Problems** section in the **Properties** pane.

Check again for problems.

- 1 In the Dependency Analyzer, in the **Properties** pane, point to the problem message, **Not in project**, under **Problems** and click the magnifying glass . The graph updates to highlight the problem file, `timesTableGame.m`.
- 2 To view the dependencies of the problem file, in the **Impact Analysis** section, click **All Dependencies**.

Now that you have seen the problem, fix it by returning the missing file to the project. Right-click the file and select **Add to Project**. The next time you run a dependency analysis, the file does not appear as a problem file.

After running a dependency analysis, to investigate the dependencies of modified files, perform an impact analysis.

- 1 In the **Views** section, click **Source Control**. The graph colors the files by source control status.
- 2 Select the modified files in the graph or in the **File List**.
- 3 To view the dependencies of the modified files, in the **Impact Analysis** section, click **All Dependencies**.

Run Project Integrity Checks

To make sure that your changes are ready to commit, check your project. On the **Project** tab, click **Run Checks** to run the project integrity checks. The checks look for missing files, files to add to source control or retrieve from source control, and other issues. The Checks dialog box offers automatic fixes to problems found, when possible. When you click a **Details** button in the Checks dialog box, you can view recommended actions and decide whether to make the changes.

Commit Modified Files

After you modify files and you are satisfied with the results of the checks, you can commit your changes to the source control repository.

- 1 In the Files view, select the **Modified (number of files)** tab. The files you changed appear in the list.
- 2 To commit your changes to source control, on the **Project** tab, in the **Source Control** section, click **Commit**.
- 3 Enter a comment for your submission, and click **Submit**. Watch the messages in the status bar as the source control commits your changes. In Git, you can have both local and remote repositories. These instructions commit to the local repository. To commit to the remote repository, in the **Source Control** section, click **Pull**.

View Project and Source Control Information

To view and edit project details, on the **Project** tab, in the **Environment** section, click **Details**. View and edit project details such as the name, description, project root, startup folder, and location of folders containing generated files.

To view details about the source control integration and repository location, on the **Project** tab, in the **Source Control** section, click **Git Details**. The Times Table App example project uses Git source control.

Close the Project

Click the  at the top right corner of the project window to close the project.

```
proj = currentProject;  
close(proj);
```

See Also

More About

- “Create Projects” on page 32-2
- “Manage Project Files” on page 32-10
- “Analyze Project Dependencies” on page 32-30

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 33-2
- “Select or Disable Source Control System” on page 33-4
- “Create New Repository” on page 33-5
- “Review Changes in Source Control” on page 33-7
- “Mark Files for Addition to Source Control” on page 33-8
- “Resolve Source Control Conflicts” on page 33-9
- “Commit Modified Files to Source Control” on page 33-12
- “Revert Changes in Source Control” on page 33-13
- “Set Up SVN Source Control” on page 33-14
- “Check Out from SVN Repository” on page 33-19
- “Update SVN File Status and Revision” on page 33-21
- “Get SVN File Locks” on page 33-22
- “Set Up Git Source Control” on page 33-23
- “Add Git Submodules” on page 33-26
- “Retrieve Files from Git Repository” on page 33-28
- “Update Git File Status and Revision” on page 33-29
- “Branch and Merge with Git” on page 33-30
- “Pull, Push and Fetch Files with Git” on page 33-34
- “Move, Rename, or Delete Files Under Source Control” on page 33-37
- “Customize External Source Control to Use MATLAB for Diff and Merge” on page 33-38
- “MSSCCI Source Control Interface” on page 33-43
- “Set Up MSSCCI Source Control” on page 33-44
- “Check Files In and Out from MSSCCI Source Control” on page 33-49
- “Additional MSSCCI Source Control Actions” on page 33-51
- “Access MSSCCI Source Control from Editors” on page 33-57
- “Troubleshoot MSSCCI Source Control Problems” on page 33-58

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

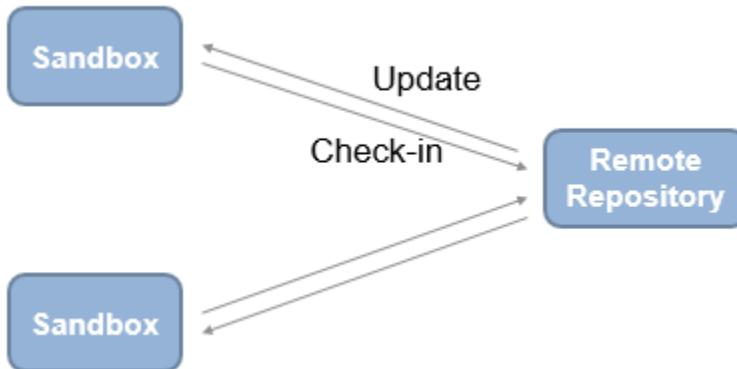
To use source control in your project, use any of these workflows:

- Retrieve files from an existing repository. See “Check Out from SVN Repository” on page 33-19 or “Retrieve Files from Git Repository” on page 33-28.
- Add source control to a folder. See “Create New Repository” on page 33-5.
- Add new files in a folder already under source control. See “Mark Files for Addition to Source Control” on page 33-8.

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 and Manage Add-Ons”.

Classic and Distributed Source Control

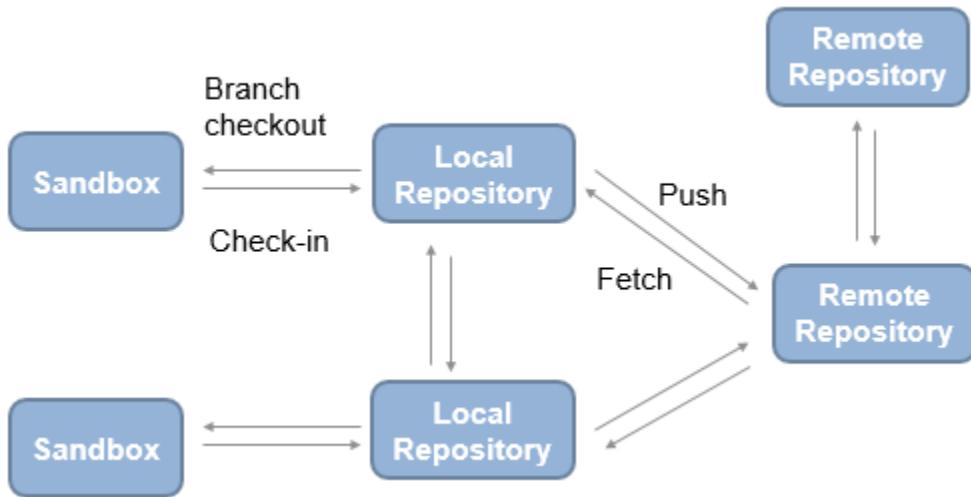
This diagram represents the classic source control workflow (for example, using SVN).



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

Create Git Repository on Your Local System

If you want to add version control to your files without sharing with another user, it is quickest to create a Git repository and sandbox on your local system. To instead clone an existing remote Git repository, see “Retrieve Files from Git Repository” on page 33-28.

Note Before using source control, you must register binary files with your source control tools to avoid corruption. For more information, see “Register Binary Files with Git” on page 33-24.

To create a Git repository and sandbox on your local system:

- 1 Right-click in the white space (any blank area) of the MATLAB Current Folder browser and select **Source Control > Manage Files**. MATLAB opens the Manage Files Using Source Control dialog box.
- 2 In the **Source control integration** list, select **Git**.
- 3 Click the **Change** button. MATLAB opens the Select a Repository dialog box.
- 4 Click the **Create a Git repository on disk** (+) button.
- 5 Select an empty folder or create a new folder in which you want to create the repository and then click **Select Folder**. MATLAB creates the repository, closes the dialog box and returns to the Select a Repository dialog box.
- 6 Click **Validate** to check the path to the new repository, and then click **OK**. MATLAB closes the dialog box and returns to the Manage Files Using Source Control dialog box.
- 7 In the **Sandbox** field, specify the location for your sandbox. The selected folder must be empty.
- 8 Click **Retrieve** to create the sandbox.

After creating the Git repository and sandbox, add your files to the sandbox. Then, commit the first version of your files to the new repository. For more information, see “Mark Files for Addition to Source Control” on page 33-8.

You also 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.

To use a Git server for the repository on your local system, you can use a Git server hosting solution or set up your own Apache™ Git server. If you cannot set up a server and must use a remote repository via the file system using the `file:///` protocol, make sure that it is a bare repository with no checked out working copy.

See Also

Related Examples

- “Set Up Git Source Control” on page 33-23
- “Set Up SVN Source Control” on page 33-14

- “Retrieve Files from Git Repository” on page 33-28
- “Check Out from SVN Repository” on page 33-19
- “Commit Modified Files to Source Control” on page 33-12

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 33-21 or “Update Git File Status and Revision” on page 33-29.

See Also

Related Examples

- “Resolve Source Control Conflicts” on page 33-9
- “Commit Modified Files to Source Control” on page 33-12
- “Revert Changes in Source Control” on page 33-13

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 33-9 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 33-10.

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 33-10.

- 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:

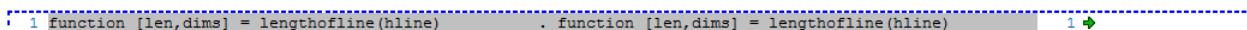
```
<<<<< .mine
```

then extract the conflict markers before merging, as described in “Extract Conflict Markers” on page 33-10.

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 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 33-10
- “Extract Conflict Markers” on page 33-11

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 33-14 or “Register Binary Files with

Git" on page 33-24. 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 33-9.

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 33-8
- “Review Changes in Source Control” on page 33-7
- “Resolve Source Control Conflicts” on page 33-9
- “Update SVN File Status and Revision” on page 33-21
- “Update Git File Status and Revision” on page 33-29
- “Pull, Push and Fetch Files with Git” on page 33-34

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 33-9

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.

SVN Source Control Options

To use the version of SVN provided with MATLAB, when you retrieve a file from source control, select SVN in the **Source control integration** list. For instructions, see “Check Out from SVN Repository” on page 33-19. 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 33-14.

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, .mlapp, .mdl, .slx, .mdlp, .slxp, .sldd, and .p 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 autmerge. 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 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 autmerge. 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 33-15.
- 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 33-15.

Create SVN Config File

- 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
*.mlapp = svn:mime-type= application/octet-stream
*.p = svn:mime-type=application/octet-stream
*.mdlp = svn:mime-type=application/octet-stream
*.slxp = svn:mime-type=application/octet-stream
*.sldd = svn:mime-type=application/octet-stream
*.slxc = svn:mime-type=application/octet-stream
*.mlproj = svn:mime-type=application/octet-stream
*.mldatx = svn:mime-type=application/octet-stream
*.slreqx = svn:mime-type=application/octet-stream
*.sfx = svn:mime-type=application/octet-stream
*.slt = svn:mime-type=application/octet-stream
```

- 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 MEX-files (.mexa64, .mexmaci64, .mexw64), .xlsx, .jpg, .pdf, .docx, etc. Add a line to the config file for each file type you need. Examples:

```
*.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
```

- 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 33-16.

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.

- Edit the config file in a text editor.
- 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  
*.mlapp = svn:mime-type= application/octet-stream  
*.p = svn:mime-type=application/octet-stream  
*.mdlp = svn:mime-type=application/octet-stream  
*.slxp = svn:mime-type=application/octet-stream  
*.sldd = svn:mime-type=application/octet-stream  
*.slxc = svn:mime-type=application/octet-stream  
*.mlproj = svn:mime-type=application/octet-stream  
*.mldatx = svn:mime-type=application/octet-stream  
*.slreqx = svn:mime-type=application/octet-stream  
*.sfx = svn:mime-type=application/octet-stream  
*.slt = 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 MEX-files (.mexa64, .mexmaci64, .mexw64), .xlsx, .jpg, .pdf, .docx, etc. Add a line to the config file for each file type you use. Examples:

```
*.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 33-16.

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 <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 33-17.

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 33-17.

- 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 33-14.

- 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 33-22.

Share a Subversion Repository

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 33-19

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 SVN.
- 3 If you know your repository location, paste it into the **Repository path** field.

Otherwise, to browse for and validate the repository path to retrieve files from, click **Change**.

- a In the Specify SVN Repository URL dialog box, specify the repository URL by entering a URL in the box, using the list of recent repositories.
- b Click **Validate** to check the repository path.

If you see an authentication dialog box for your repository, enter the login information to continue.

- c If the path is invalid, check the URL against your source control repository browser.

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.

- d When you have finished specifying the URL path you want to retrieve, click **OK**. The dialog box closes and you return to the Manage Files Using Source Control dialog box.
- 4 In the Manage Files Using Source Control dialog box, select the sandbox folder to store the retrieved files and click **Retrieve**.

If you see an authentication dialog box for your repository, enter the login information to continue.

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 33-21.

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 33-17.

- 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 SVN.
- 3 Click **Change** to select the repository path that you want to retrieve files from.
- 4 In the Specify SVN Repository URL dialog box:
 - a Select a recent repository from the **Repository** list, or click the **Repository** 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 33-14
- “Update SVN File Status and Revision” on page 33-21

Update SVN File Status and Revision

In this section...

["Refresh Status of Files" on page 33-21](#)

["Update Revisions of Files" on page 33-21](#)

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 33-21](#).

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 33-19](#)
- ["Review Changes in Source Control" on page 33-7](#)

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**.

Note Starting in R2020a Update 5, SVN cleanup only removes stale locks and unfinished transactions. It does not remove unversioned or ignored files.

You can manually remove unversioned and ignored files.

- 1 In the Current Folder browser, click the **SVN** header to sort files by their SVN status.
 - 2 Select the **Not Under Source Control** files.
 - 3 Right-click and select **Delete**.
-

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 33-17

Set Up Git Source Control

In this section...

- “Configure MATLAB on Windows” on page 33-23
- “Use SSH Authentication with MATLAB” on page 33-23
- “Register Binary Files with Git” on page 33-24
- “Configure Git Credential Helper” on page 33-25

Note Starting in R2020b, you do not need to install command-line Git to fully use Git with MATLAB. You can now merge branches using the built-in Git integration.

To setup Git for releases before R2020b, see https://www.mathworks.com/help/releases/R2020a/matlab/matlab_prog/set-up-git-source-control.html.

Configure MATLAB on Windows

Several operations, such as committing, merging, and receiving pushed commits, use Git Hooks. To use Git Hooks on Windows with MATLAB, install Cygwin and add it to the MATLAB library path:

- 1 Download the installer from <https://www.cygwin.com/>. Run the installer.
- 2 In the MATLAB Command Window, type
`edit(fullfile(matlabroot,"toolbox","local","librarypath.txt")).`

Add the Cygwin bin folder location to the end of `librarypath.txt`, for example,
`C:\cygwin64\bin`.

If you do not have permission to edit the `librarypath.txt` file, see “Locate Native Method Libraries”.

- 3 Restart MATLAB for the changes to take effect.

You can clone a remote repository like GitHub and GitLab using HTTPS or SSH. To prevent frequent login prompts when you interact with your remote repository using HTTPS, configure a Git credential manager to remember credentials or add a new public key and clone the repository using SSH instead. For more information, see “Use SSH Authentication with MATLAB” on page 33-23.

For new projects under Git source control, MATLAB automatically registers your binary files to prevent corruption when merging. For existing projects, register the binary files before using Git to merge branches. For more information, see “Register Binary Files with Git” on page 33-24.

If you are working with long path files, run this command in MATLAB:

```
!git config --global core.longpaths true
```

Use SSH Authentication with MATLAB

To prevent frequent login prompts when you interact with your remote repository using HTTPS, add a new public key and clone the repository using SSH instead.

To use SSH authentication inside MATLAB:

- 1 Use `ssh-keygen` to generate valid SSH keys. In the Command Prompt, enter:

```
ssh-keygen
```

```
Generating public/private rsa key pair.  
Enter file in which to save the key (C:\Users\username/.ssh/id_rsa):  
Created directory 'C:\Users\username/.ssh'.  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in C:\Users\username/.ssh/id_rsa.  
Your public key has been saved in C:\Users\username/.ssh/id_rsa.pub.  
The key fingerprint is:  
SHA256:/Nc9/tnZ7Dmh77+iJMxmPVrlPqaFd6J1j1YRXEk3Tgs company\username@us-username
```

`ssh-keygen` confirms where to save the key (for example, `.ssh/id_rsa`) and asks for a passphrase. If you do not want to type a password when you use the key, leave the passphrase empty. If you already have keys in the specified folder, `ssh-keygen` asks if you want to override them.

Note It is not possible to generate SSH keys directly in MATLAB. Generate SSH keys using the `ssh-keygen` provided with a command-line Git install.

- 2 On Linux and Mac, place your keys in the `HOME/.ssh` folder. To verify which `HOME` directory the MATLAB Git integration is working with, in the MATLAB Command Window, enter:

```
getenv('HOME')
```

To use multiple keys or keys with pass-phrases, use an SSH agent. If an SSH agent is running, MATLAB looks for keys in the agent before looking in `HOME/.ssh`.

On Windows, place your keys in the `USERPROFILE/.ssh` or `HOME/.ssh` folder. To verify which `USERPROFILE` directory the MATLAB Git integration is working with, in the MATLAB Command Window, enter:

```
getenv('USERPROFILE')
```

To use multiple keys or keys with pass-phrases, use Pageant as the SSH agent. If Pageant is running, MATLAB looks for keys in Pageant before looking in `USERPROFILE/.ssh`.

- 3 Configure your GitHub or GitLab account to use the SSH keys:

- Copy the contents of `.pub` file in the `.ssh` folder.
- Paste the contents in the Add SSH key field in the SSH keys section of your account settings.

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`, `.mlapp`, `.mdl`, `.slx`, `.mdlp`, `.slxp`, `.sldd`, and `.p` 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 autmerge. 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 MEX-files, `.xlsx`, `.jpg`, `.pdf`, `.docx`, etc.

You can prevent Git from corrupting your files by registering binary files in your `.gitattributes` file.

- For new projects and projects that switched from another source control system, MATLAB automatically creates a `.gitattributes` file and populates it with a list of binary files to register. This specifies that Git should not make automatic line feed, diff, and merge attempts for registered files.
- For existing projects, create a `.gitattributes` file and populate it with the list of binary files to register.

- 1 In the Command Window, type:

```
edit .gitattributes
```

- 2 Add a line to the attributes file for each file type you need. For example, `*.mlapp binary`.

Tip You can copy a `.gitattributes` file that contains the list of common binary files to register.

```
copyfile(fullfile(matlabroot, 'toolbox', 'shared', 'cmlink', 'git', 'auxiliary_files', 'mwgitatt...
```

- 3 Restart MATLAB so you can start using the Git client.

Tip You can reduce your Git repository size by saving Simulink models without compression. Turning off compression results in larger SLX files on disk but reduces repository size.

To use this setting with new SLX files, create your models using a model template with SLX Compression set to none. For existing SLX files, set compression and then save the model. For more information, see “Set SLX Compression Level” (Simulink).

Configure Git Credential Helper

You can configure Git to use a credential helper to remember usernames and passwords. For all platforms, Git Credential Manager Core is recommended as the credential helper.

To remember credentials, Git must be installed. On Windows, install Git for Windows.

See Also

Related Examples

- “Clone from Git Repository” on page 32-44

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**.

Update Submodules

After using **Pull** to get the latest changes from a remote repository, check that submodules are up to date by clicking Submodules and then click **Update**. If any submodule definition have changed, then the update ensures that the submodule folder contains the correct files. Update applies to all child submodules in the submodule hierarchy.

Use Fetch and Merge with Submodules

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 33-35.

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

- “Retrieve Files from Git Repository” on page 33-28

Retrieve Files from Git Repository

You can use source control with projects. For more information, see “Clone from Git Repository” on page 32-44 and “Use Source Control with Projects” on page 32-45.

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 If you know your repository location, paste it into the **Repository path** field.

Otherwise, to browse for and validate the repository path to retrieve files from, click **Change**.

- a In the Select a Repository dialog box, browse to your repository using the **Remote** button .
- b Click **Validate** to check the repository path.

If you see an authentication dialog box for your repository, enter the login information to continue.

- c If the path is valid, click **OK**. The dialog box closes and you return to the Manage Files Using Source Control dialog box.

- 4 In the Manage Files Using Source Control dialog box, select the sandbox folder to store the retrieved files and click **Retrieve**.

If you see an authentication dialog box for your repository, enter the login information to continue.

See Also

Related Examples

- “Set Up Git Source Control” on page 33-23
- “Update Git File Status and Revision” on page 33-29
- “Branch and Merge with Git” on page 33-30

Update Git File Status and Revision

In this section...

"Refresh Status of Files" on page 33-29

"Update Revisions of Files" on page 33-29

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 33-24.

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 33-34.

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

- "Retrieve Files from Git Repository" on page 33-28
- "Review Changes in Source Control" on page 33-7

Branch and Merge with Git

In this section...

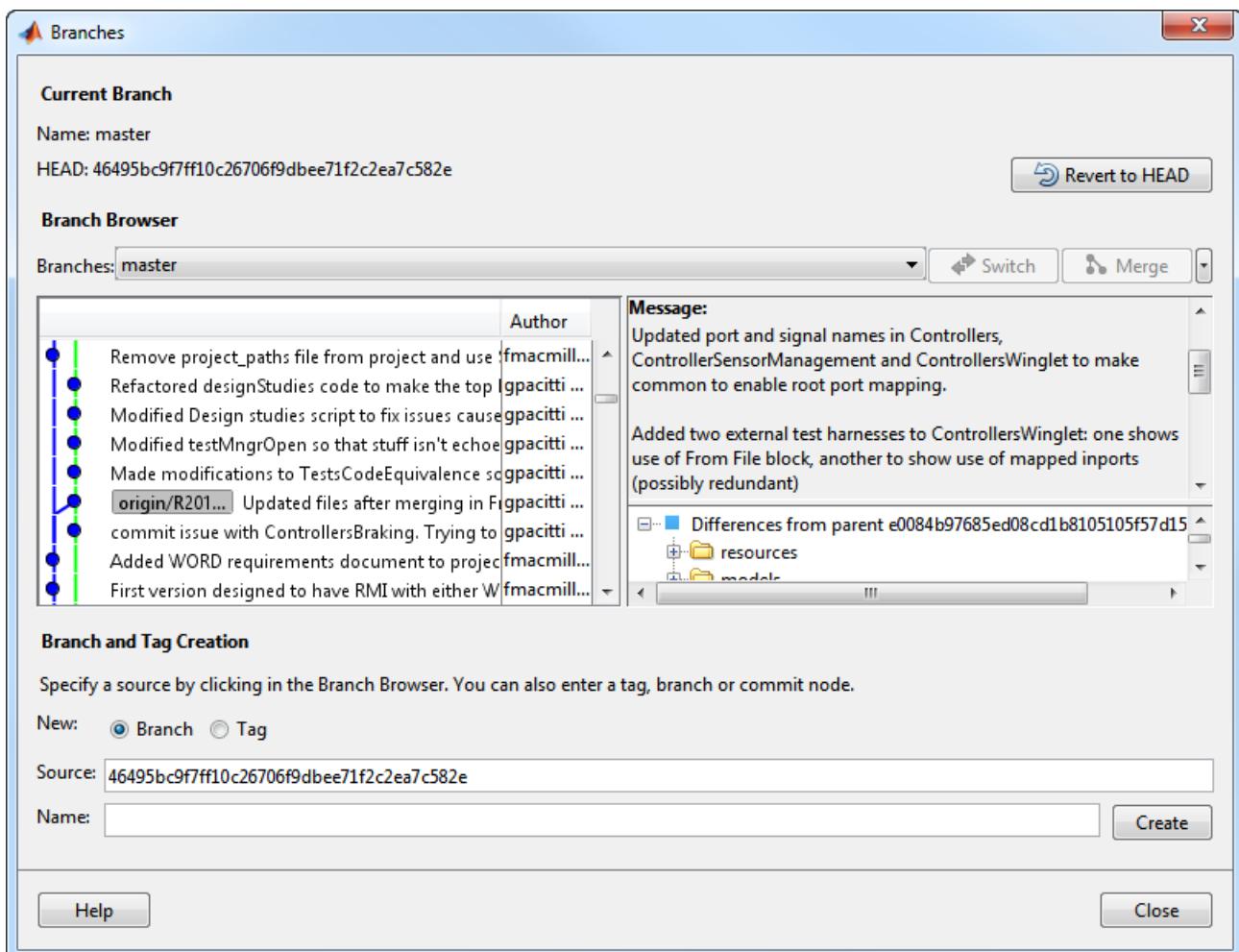
- “Create Branch” on page 33-30
- “Switch Branch” on page 33-31
- “Compare Branches” on page 33-32
- “Merge Branches” on page 33-32
- “Revert to Head” on page 33-33
- “Delete Branches” on page 33-33

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 33-34.

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**.

- To examine differences in a file between the current revision and its parent, right-click a file in the tree under **Differences from parent** and select **Show Difference**.
- To examine differences in a file between any two revisions including revisions on two different development branches, hold the **Ctrl** key and select the two different revisions. Right-click a file in the tree under **Differences from selection** and select **Show Difference**.

MATLAB opens a comparison report. You can save a copy of the selected file on either revision. Right-click a file and select **Save As** to save a copy of the file on the selected revision. Select **Save Original As** to save a copy of the file on the prior revision. This is useful if you want to test how the code ran in previous revisions or on other branches.

Merge Branches

Before you can merge branches, you must register binary files to prevent Git from inserting conflict markers. See “Register Binary Files with Git” on page 33-24.

Tip After you fetch changes, you must merge. For more information, see “Fetch and Merge” on page 33-35.

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 33-9.

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.

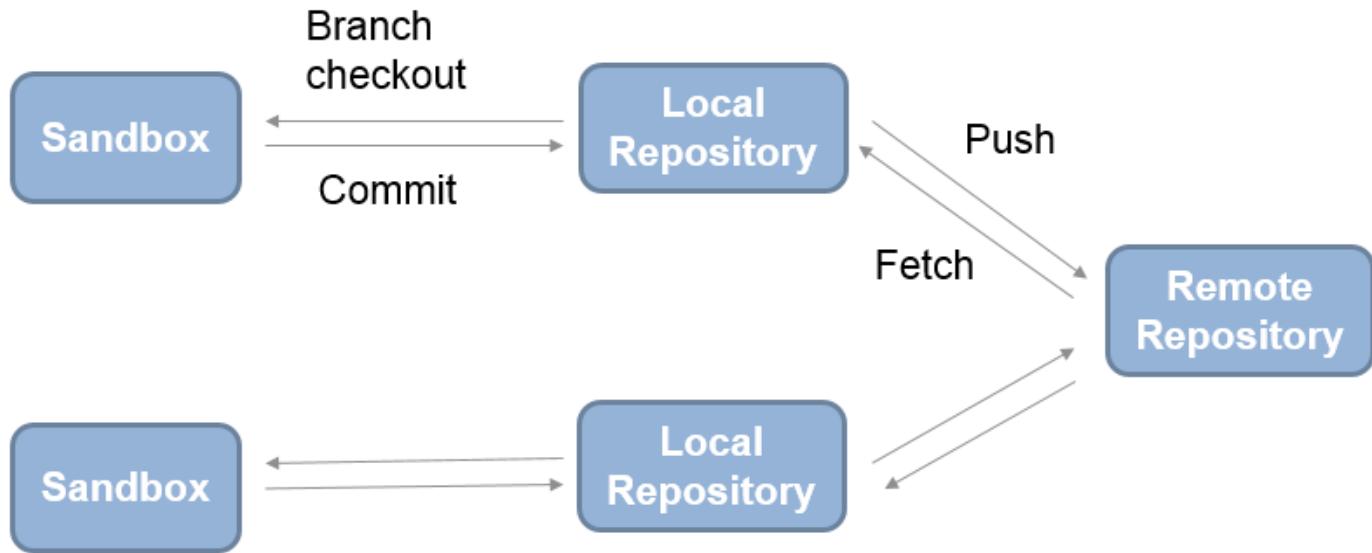
See Also

Related Examples

- “Set Up Git Source Control” on page 33-23
- “Pull, Push and Fetch Files with Git” on page 33-34
- “Resolve Source Control Conflicts” on page 33-9

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 register binary files to prevent Git from inserting conflict markers. See “Register Binary Files with Git” on page 33-24.

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 register binary files to prevent Git from inserting conflict markers. See “Register Binary Files with Git” on page 33-24.

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 33-30
- “Resolve Source Control Conflicts” on page 33-9

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 33-8
- “Commit Modified Files to Source Control” on page 33-12

Customize External Source Control to Use MATLAB for Diff and Merge

In this section...

"[Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge](#)" on page 33-38

"[Integration with Git](#)" on page 33-39

"[Integration with SVN](#)" on page 33-40

"[Integration with Other Source Control Tools](#)" on page 33-41

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. The MATLAB Comparison Tool provides tools for merging MathWorks files and is compatible with popular software configuration management and version control systems. You can use the automerge tool with Git to automatically merge branches that contain changes in different subsystems in the same SLX file.

To set up your source control tool to use MATLAB as the application for diff and merge, you must first determine the full paths of the `mlDiff`, `mlMerge`, and `mlAutoMerge` executable files, and then follow the recommended steps for the source control tool you are using.

Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge

To get the required file paths and enable external source control tools 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**.

This command also displays the file paths to copy and paste into your source control tool setup:

- On Windows:

Diff: `matlabroot\bin\win64\mlDiff.exe`

Merge: `matlabroot\bin\win64\mlMerge.exe`

AutoMerge: `matlabroot\bin\win64\mlAutoMerge.exe`

- On Linux:

Diff: `matlabroot/bin/glnxa64/mlDiff`

Merge: `matlabroot/bin/glnxa64/mlMerge`

AutoMerge: `matlabroot/bin/glnxa64/mlAutoMerge`

- On Mac:

Diff: `matlabroot/bin/maci64/mlDiff`

Merge: `matlabroot/bin/maci64/mlMerge`

AutoMerge: `matlabroot/bin/maci64/mlAutoMerge`

where `matlabroot` is replaced with the full path to your installation, for example, `C:\Program Files\MATLAB\R2020b`.

Note Your diff and merge operations use open MATLAB sessions when available, and only open MATLAB when necessary. The operations only use the specified MATLAB installation.

Integration with Git

Command Line

To configure MATLAB diff and merge tools with command-line Git:

- 1 Run this command in MATLAB.

```
comparisons.ExternalSCMLink.setupGitConfig()
```

This command displays the full paths of the `mlDiff`, `mlMerge`, and `mlAutoMerge` executable files. It also automatically populates the global `.gitconfig` file. For example:

```
[difftool "mlDiff"]
cmd = \"/C:/Program Files/MATLAB/R2020b/bin/win64/mlDiff.exe\" $LOCAL $REMOTE
[mergetool "mlMerge"]
cmd = \"/C:/Program Files/MATLAB/R2020b/bin/win64/mlMerge.exe\" $BASE $LOCAL $REMOTE $MERGED
[merge "mlAutoMerge"]
driver = \"/C:/Program Files/MATLAB/R2020b/bin/win64/mlAutoMerge.exe\" %0 %A %B %A
```

Note You need to do step 1 only once for your Git setup.

- 2 Configure your repository to use the `mlAutoMerge` executable file. Open the `.gitattributes` file in your repository and add:

```
*.slx binary merge=mlAutoMerge
```

Now, when you merge branches that contain changes in different subsystems in the same SLX file, MATLAB handles the merge automatically.

To run the MATLAB diff and merge tools from command-line Git, use `git difftool` and `git mergetool`:

- To compare two revisions of a model using the MATLAB diff tool, type:

```
git difftool -t mlDiff <revisionID1> <revisionID2> myModel.slx
```

If you do not provide revision IDs, `git difftool` compares the working copy to the repository copy.

If you do not specify which model you want to compare, command-line Git will go through all modified files and ask you if you want to compare them one by one.

- To resolve a merge conflict in a model using the MATLAB merge tool, type:

```
git mergetool -t mlMerge myModel.slx
```

If you do not specify which model you want to merge, command-line Git will go through all files and ask you if you want to merge them one by one.

SourceTree

SourceTree is an interactive GUI tool that visualizes and manages Git repositories for Windows and Mac.

- 1 Configure the MATLAB diff and merge tools as SourceTree external tools:
 - a With SourceTree open, click **Tools > Options**.
 - b On the **Diff** tab, under **External Diff / Merge**, fill the fields with the following information:

```
External Diff tool: Custom
Diff Command: C:\Program Files\MATLAB\R2020b\bin\win64\mlDiff.exe
Arguments: $LOCAL $REMOTE
Merge tool: Custom
Merge Command: C:\Program Files\MATLAB\R2020b\bin\win64\mlMerge.exe
Arguments: $BASE $LOCAL $REMOTE $MERGED
```
- 2 Configure your repository to automerge changes in different subsystems in the same SLX file using the **mlAutoMerge** executable file:
 - a Open the global **.gitconfig** file and add:

```
[merge "mlAutoMerge"]
driver = "C:/Program Files/MATLAB/R2020b/bin/win64/mlAutoMerge.exe" %0 %A %B %A
```
 - b Open the **.gitattributes** file in your repository and add:

```
*.slx binary merge=mlAutoMerge
```

Tip Customize the full path of the **mlDiff**, **mlMerge**, and **mlAutoMerge** executables to match both the MATLAB installation and the operating system you are using. For more information, see “[Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge](#)” on page 33-38.

To use the MATLAB diff tool from within SourceTree, right-click a modified file under **Unstaged files** and select **External Diff**.

To use the MATLAB merge tool when SourceTree detects a merge conflict, select the **Uncommitted changes** branch, right-click a modified file, and select **Resolve Conflicts > Launch External Merge Tool**.

Integration with SVN

TortoiseSVN

With TortoiseSVN, you can customize your diff and merge tools based on the file extension. For example, to use MATLAB diff and merge tools for SLX files:

- 1 Right-click in any file explorer window and select **TortoiseSVN > Settings** to open TortoiseSVN settings.
- 2 In the **Settings** sidebar, select **Diff Viewer**. Click **Advanced** to specify the diff application based on file extensions.
- 3 Click **Add** and fill the fields with the extension and the **mlDiff** executable path:

```
Filename, extension or mime-type: .slx
External Program: "C:\Program Files\MATLAB\R2020b\bin\win64\mlDiff.exe" %base %mine
```
- 4 Click **OK** and repeat the same steps to add another file extension.

- 5 In the **Settings** sidebar, select **Diff ViewerMerge Tool**. Click **Advanced** to specify the merge application based on file extensions.

- 6 Click **Add** and fill the fields with the extension and `mlMerge` executable path:

```
Filename, extension or mime-type: .slx
External Program: "C:\Program Files\MATLAB\R2020b\bin\win64\mlMerge.exe" %base %mine %theirs %merged
```

- 7 Click **OK** and repeat the same steps to add another file extension.

You can now use the MATLAB tools for diff and merge the same way you would use the TortoiseSVN default diff and merge applications.

Note Automerging binary files with SVN , such as SLX files, is not supported.

Integration with Other Source Control Tools

Perforce P4V

With Perforce® P4V, you can customize your diff and merge tools based on the file extension. To use MATLAB diff and merge tools for SLX files, for example:

- 1 In Perforce, click **Edit > Preferences**.
- 2 In the **Preferences** sidebar, select **Diff**. Under **Specify diff application by extension (overrides default)**, click **Add**.
- 3 In the **Add File Type** dialog box, enter the following information:

```
Extension: .slx
Application: C:\Program Files\MATLAB\R2020b\bin\win64\mlDiff.exe
Arguments: %1 %2
```

- 4 Click **Save**.
- 5 In the **Preferences** sidebar, select **Merge**. Under **Specify merge application by extension (overrides default)**, click **Add**.
- 6 In the **Add File Type** dialog box, enter the following information:

```
Extension: .slx
Application: C:\Program Files\MATLAB\R2020b\bin\win64\mlMerge.exe
Arguments: %b %2 %1 %r
```

- 7 Click **Save** and repeat the steps for other file extensions.

Tip Customize the full path of the `mlDiff` and `mlMerge` executables to match both the MATLAB installation and the operating system you are using. For more information, see “Finding the Full Paths for MATLAB Diff, Merge, and AutoMerge” on page 33-38.

You can now use the MATLAB tools for diff and merge the same way you would use the Perforce default diff and merge applications.

See Also

Related Examples

- “Compare Files and Folders and Merge Files”

- “Compare and Merge MAT-Files”
- “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 33-57.

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 33-44](#)

[“Specify Source Control System with MATLAB Software” on page 33-45](#)

[“Register Source Control Project with MATLAB Software” on page 33-46](#)

[“Add Files to Source Control” on page 33-48](#)

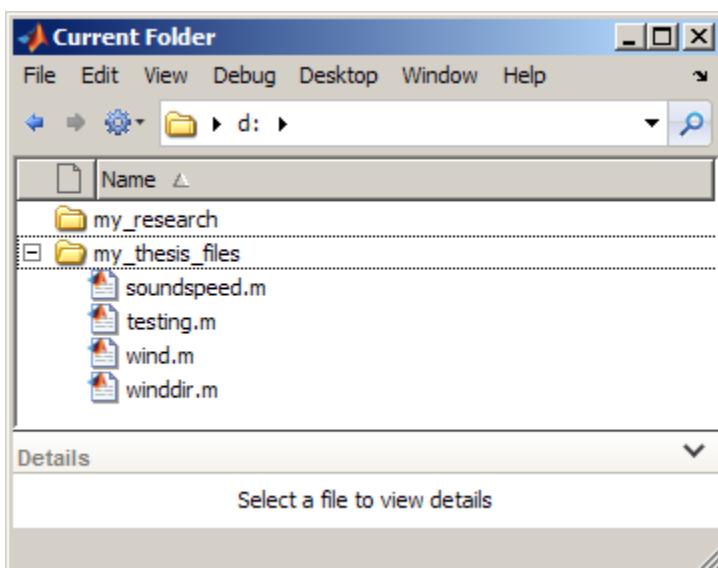
Create Projects in Source Control System

In your source control system, create the projects that your folders and files will be associated with.

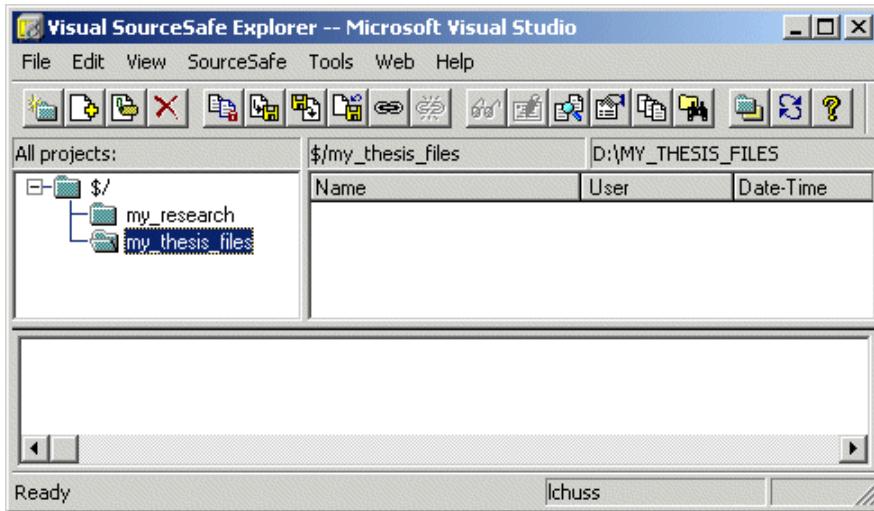
All files in a folder must belong to the same source control project. Be sure the working folder for the project in the source control system specifies the correct path to the folder on disk.

Example of Creating Source Control Project

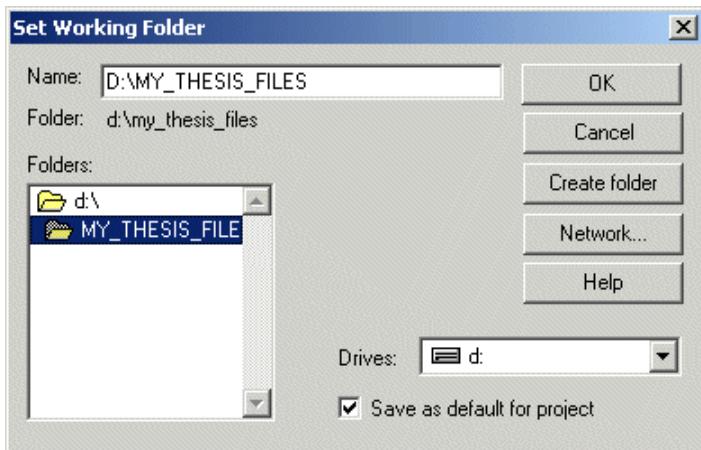
This example uses the project `my_thesis_files` in Microsoft Visual SourceSafe. This illustration of the Current Folder browser shows the path to the folder on disk, `D:\my_thesis_files`.



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.

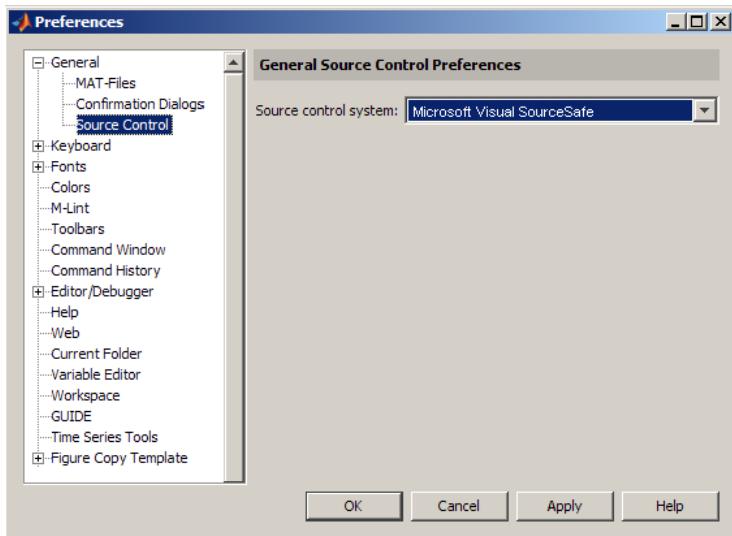


Specify Source Control System with MATLAB Software

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

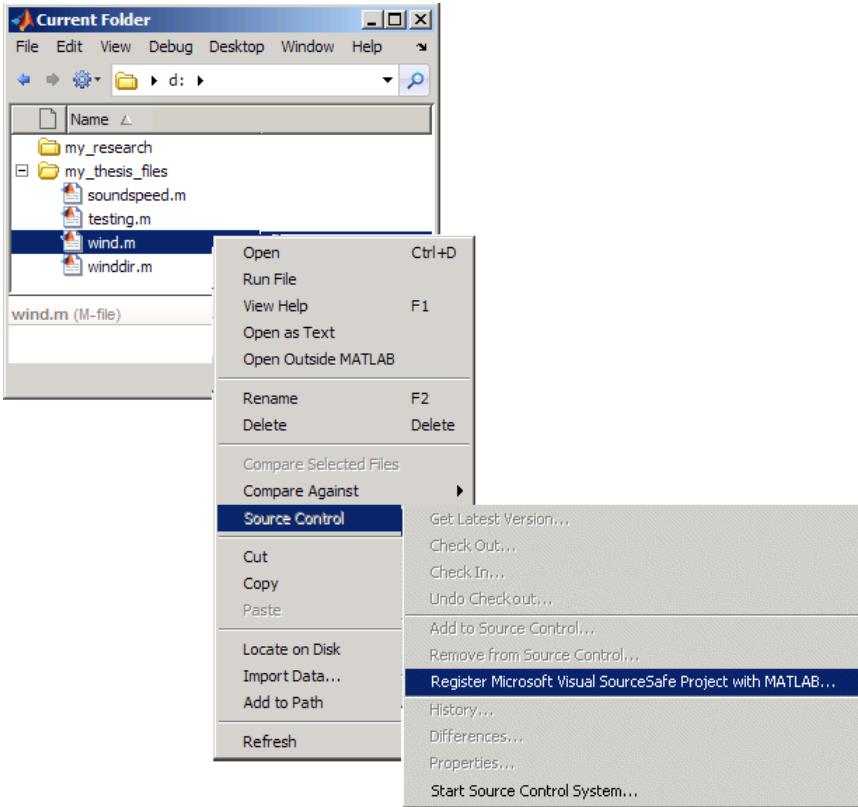
If you run a 64-bit version of MATLAB and want MATLAB to interface with your source control system, your source control system must be 64-bit compliant. If you have a 32-bit source control system, or if you have a 64-bit source control system running in 32-bit compatibility mode, MATLAB cannot use it. 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 33-45.

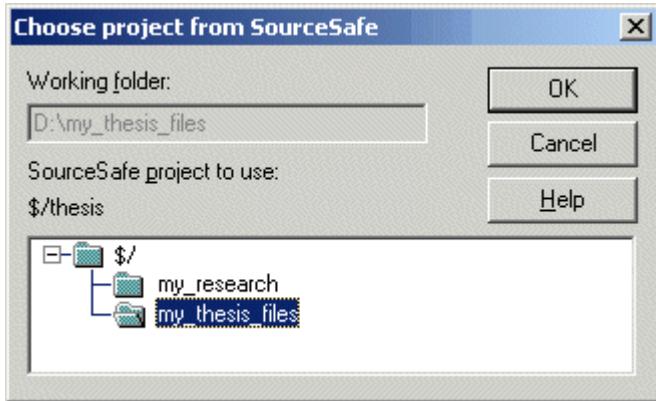
The following example shows Microsoft Visual SourceSafe.



- 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**.

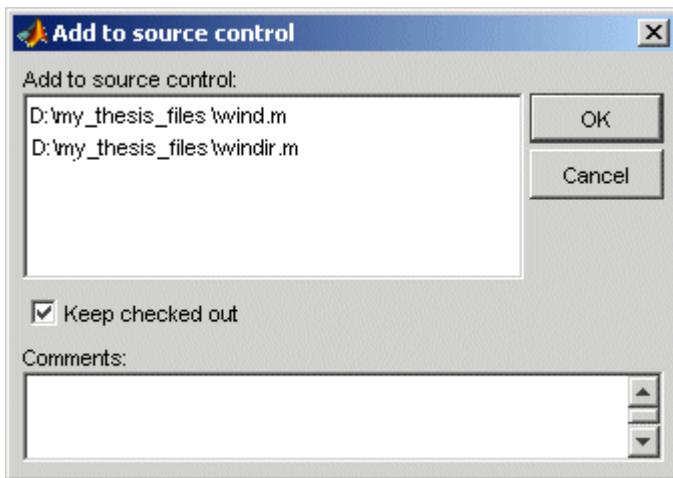


The selected file, its folder, and all files in the folder, are associated with the source control system project you selected. For the example, MATLAB associates all files in D:\my_thesis_files with the source control project my_thesis_files.

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 33-49
 “Check Files Out of Source Control” on page 33-49
 “Undoing the Checkout” on page 33-50

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 33-44.

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:

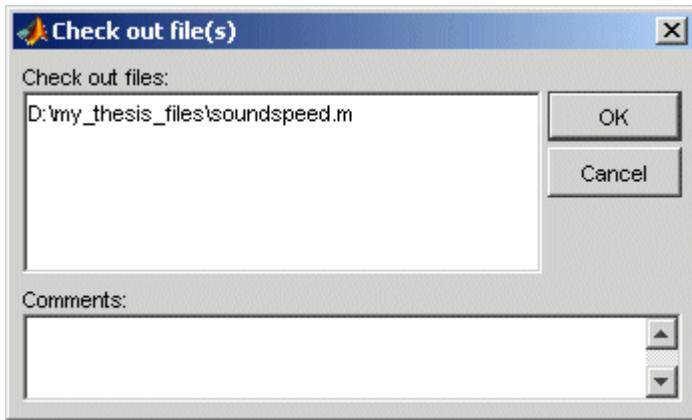
- 1 In the Current Folder browser, select the files to check in. A file can be open or closed when you check it in, but it must be saved, that is, it cannot contain unsaved changes.
- 2 Right-click, and from the context menu, select **Source Control > Check In**.
- 3 In the resulting **Check in file(s)** 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.

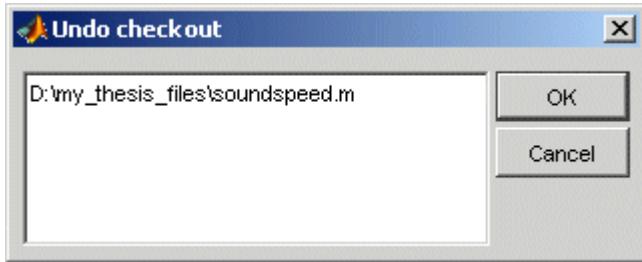
Undoing the Checkout

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 33-51
- “Removing Files from the Source Control System” on page 33-52
- “Showing File History” on page 33-52
- “Comparing the Working Copy of a File to the Latest Version in Source Control” on page 33-53
- “Viewing Source Control Properties of a File” on page 33-54
- “Starting the Source Control System” on page 33-55

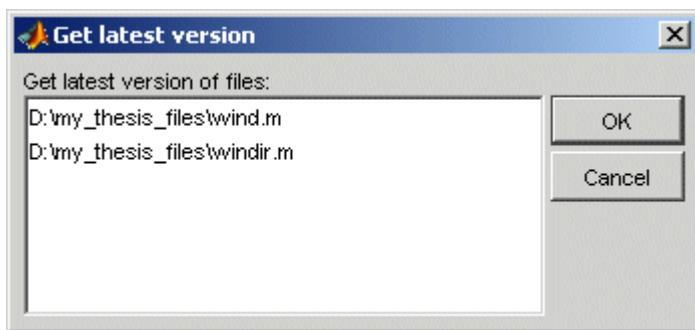
Getting the Latest Version of Files for Viewing or Compiling

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.



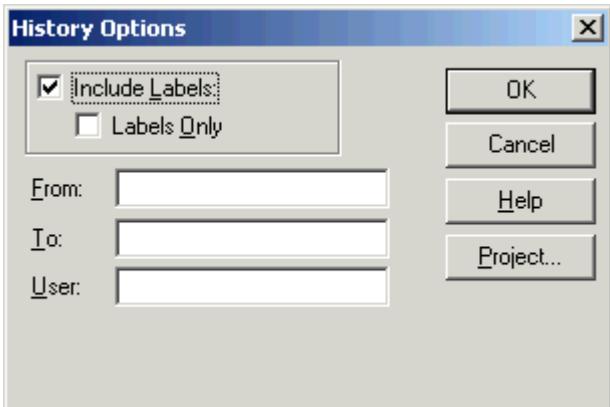
- 3 Click **OK**.

Showing File History

To show the history of a file in the source control system, follow these steps:

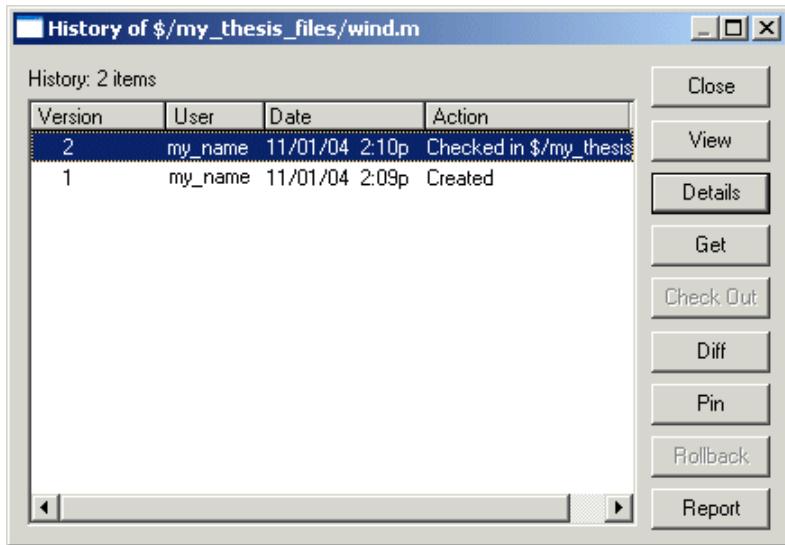
- 1 In the MATLAB Current Folder browser, select the file for which you want to view the history.
- 2 Right-click, and from the context menu, select **Source Control > History**.

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

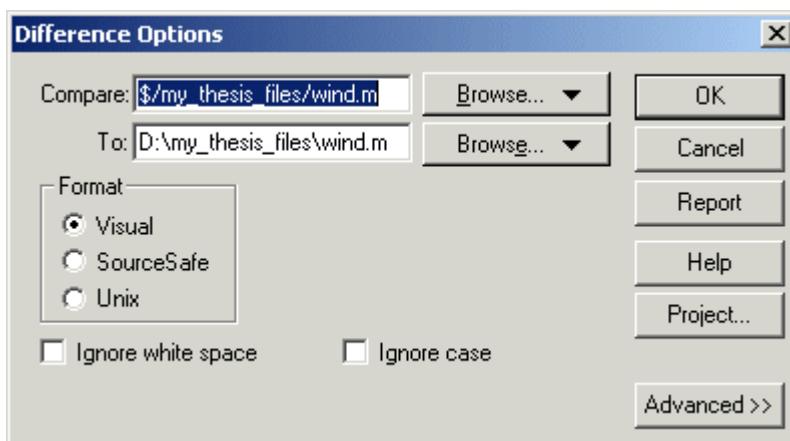
You can compare the current working copy of a file with the latest checked-in version of the file in the source control system. This highlights the differences between the two files, showing the changes you made since you checked out the file.

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.

```

Differences for $/my_thesis_files/wind.m
$:/my_thesis_files/wind.m          D:\my_thesis_files\wind.m
1 % wind speed                   1 % wind speed
2 load wdir                      2 load wdir
3 wdir                           3 wdir
4 load knots                     4 load knots
5 knots                          5 knots
6 rdir = wdir * pi/180;          6 rdir = wdir * pi/180;
7 [x,y] = pol2cart(rdir, kno    7 [x,y] = pol2cart(rdir, kno
8 compass(x,y)                  8 compass(x,y)
9                                9
10 % simulation
11 % include new code here.

```

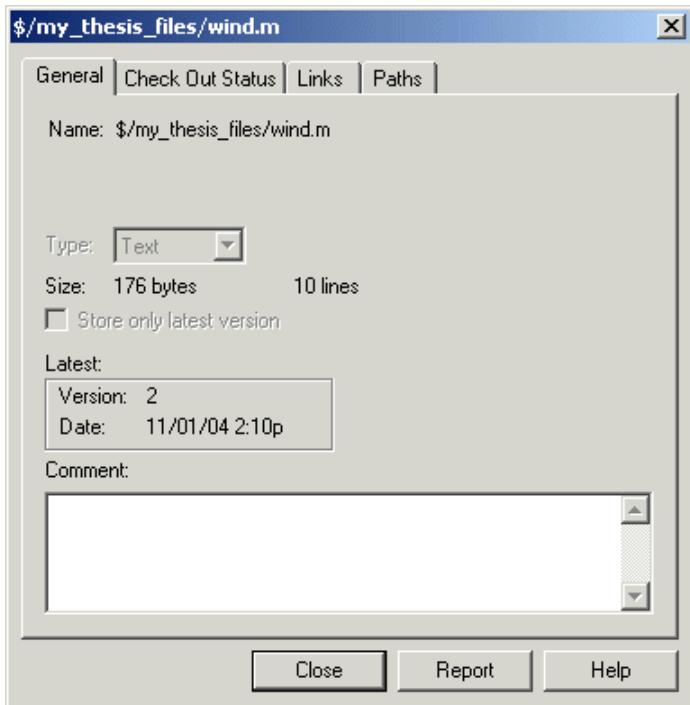
Deleted lines | Changed lines | Inserted lines | Ln 9, Col 1

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.

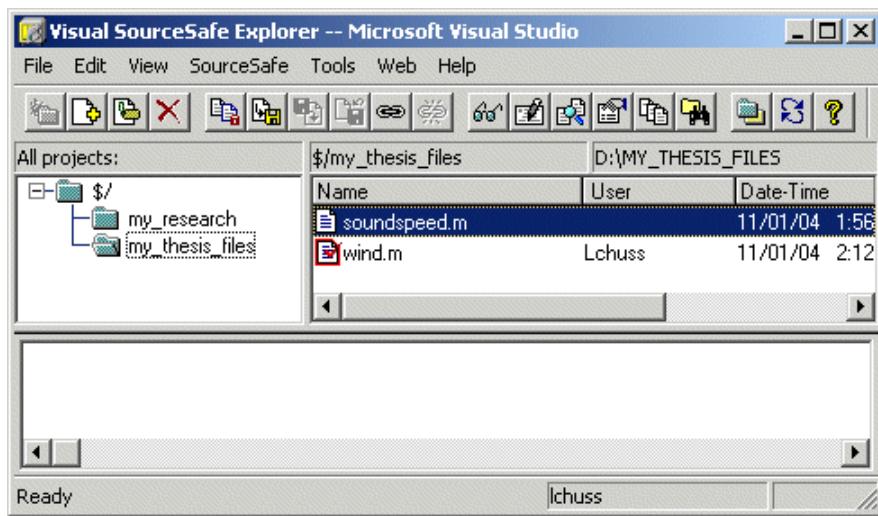


Starting the Source Control System

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 33-46.

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 33-58](#)

[“Restriction Against @ Character” on page 33-59](#)

[“Add to Source Control Is the Only Action Available” on page 33-59](#)

[“More Solutions for Source Control Problems” on page 33-59](#)

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\SccServerPath`

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 33-46. 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 34-3
- “Write Script-Based Unit Tests” on page 34-6
- “Write Script-Based Test Using Local Functions” on page 34-11
- “Extend Script-Based Tests” on page 34-14
- “Run Tests in Editor” on page 34-17
- “Write Function-Based Unit Tests” on page 34-20
- “Write Simple Test Case Using Functions” on page 34-24
- “Write Test Using Setup and Teardown Functions” on page 34-27
- “Extend Function-Based Tests” on page 34-32
- “Author Class-Based Unit Tests in MATLAB” on page 34-36
- “Write Simple Test Case Using Classes” on page 34-39
- “Write Setup and Teardown Code Using Classes” on page 34-42
- “Table of Verifications, Assertions, and Other Qualifications” on page 34-45
- “Tag Unit Tests” on page 34-47
- “Write Tests Using Shared Fixtures” on page 34-51
- “Create Basic Custom Fixture” on page 34-54
- “Create Advanced Custom Fixture” on page 34-56
- “Use Parameters in Class-Based Tests” on page 34-61
- “Create Basic Parameterized Test” on page 34-66
- “Create Advanced Parameterized Test” on page 34-71
- “Use External Parameters in Parameterized Test” on page 34-78
- “Define Parameters at Suite Creation Time” on page 34-82
- “Create Simple Test Suites” on page 34-89
- “Run Tests for Various Workflows” on page 34-91
- “Programmatically Access Test Diagnostics” on page 34-94
- “Add Plugin to Test Runner” on page 34-95
- “Write Plugins to Extend TestRunner” on page 34-97
- “Create Custom Plugin” on page 34-100
- “Run Tests in Parallel with Custom Plugin” on page 34-105
- “Write Plugin to Add Data to Test Results” on page 34-113
- “Write Plugin to Save Diagnostic Details” on page 34-118
- “Plugin to Generate Custom Test Output Format” on page 34-122
- “Analyze Test Case Results” on page 34-125
- “Analyze Failed Test Results” on page 34-128
- “Rerun Failed Tests” on page 34-130

- “Dynamically Filtered Tests” on page 34-133
- “Create Custom Constraint” on page 34-139
- “Create Custom Boolean Constraint” on page 34-142
- “Create Custom Tolerance” on page 34-146
- “Overview of App Testing Framework” on page 34-150
- “Write Test for App” on page 34-155
- “Write Test That Uses App Testing and Mocking Frameworks” on page 34-159
- “Overview of Performance Testing Framework” on page 34-164
- “Test Performance Using Scripts or Functions” on page 34-168
- “Test Performance Using Classes” on page 34-172
- “Measure Fast Executing Test Code” on page 34-178
- “Create Mock Object” on page 34-181
- “Specify Mock Object Behavior” on page 34-188
- “Qualify Mock Object Interaction” on page 34-193
- “Ways to Write Unit Tests” on page 34-199
- “Compile MATLAB Unit Tests” on page 34-202
- “Develop and Integrate Software with Continuous Integration” on page 34-205
- “Generate Artifacts Using MATLAB Unit Test Plugins” on page 34-209
- “Continuous Integration with MATLAB on CI Platforms” on page 34-213

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 test file must start or end with the word 'test', which is case-insensitive. If the file name does not start or end with the word 'test', the tests in the file might be ignored in certain cases.
- Place each unit test into a separate section of the live-script file. If a section has a heading in the Heading 1 style, the heading becomes the name of the test element. Otherwise, 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.

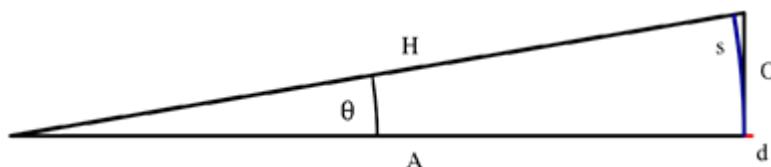
```
type rightTri.m
function angles = rightTri(sides)
A = atand(sides(1)/sides(2));
B = atand(sides(2)/sides(1));
hypotenuse = sides(1)/sind(A);
C = asind(hypotenuse*sind(A)/sides(1));

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

$$\sum_k a_k = 180^\circ$$

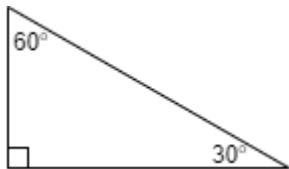
You can have multiple `assert` statements in the same test. However, if the first assertion fails, the MATLAB does not evaluate the remaining statements.

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 34-6
- “Write Function-Based Unit Tests” on page 34-20

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 test file must start or end with the word 'test', which is case-insensitive. If the file name does not start or end with the word 'test', the tests in the file might be ignored in certain cases.
- 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 =
```

```
4×6 table
```

Name	Passed	Failed	Incomplete	Duration
{'rightTriTolTest/Test1_SumOfAngles'}	true	false	false	0.02
{'rightTriTolTest/Test2_IsoscelesTriangles'}	true	false	false	0.0047
{'rightTriTolTest/Test3_30_60_90Triangle'}	true	false	false	0.0053
{'rightTriTolTest/Test4_SmallAngleApproximation'}	true	false	false	0.0049

See Also

`assert` | `runtests`

Related Examples

- “Write Script-Based Test Using Local Functions” on page 34-11
- “Write Function-Based Unit Tests” on page 34-20

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 $\frac{\pi}{4}$ are equal within a relative tolerance of 0.1%.
- Test `matches MATLAB fcn` tests whether the computed sine and cosine of $\frac{2\pi}{3}$ 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	0.39568
{'approxSinCosTest/Test2pi'}	true	false	false	0.023482
{'approxSinCosTest/TestPiOver4Equality'}	true	false	false	0.037059
{'approxSinCosTest/TestMatchesMATLABFcn'}	true	false	false	0.050943

See Also

[assert](#) | [runtests](#)

Related Examples

- “Write Script-Based Unit Tests” on page 34-6

More About

- “Add Functions to Scripts” on page 18-14

Extend Script-Based Tests

In this section...

- “Test Suite Creation” on page 34-14
- “Test Selection” on page 34-14
- “Programmatic Access of Test Diagnostics” on page 34-15
- “Test Runner Customization” on page 34-15

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 `matlab.unittest.TestSuite.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 `matlab.unittest.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 `matlab.unittest.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 `matlab.unittest.TestSuite.fromFile`.
- Use a `selectors` instance and optional `constraints` instance.

Use these approaches in a suite generation method, such as `matlab.unittest.TestSuite.fromFile`, or create a suite and filter it using the `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 `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

In certain cases, the testing framework uses a `DiagnosticsRecordingPlugin` plugin to record diagnostics on test results. The framework uses the plugin by default if you do any of these:

- Run tests using the `runtests` function.
- Run tests using the `testrunner` function with no input.
- Run tests using the `run` method of the `TestSuite` or `TestCase` classes.
- Run performance tests using the `runperf` function.
- Run performance tests using the `run` method of the `TimeExperiment` class.

After you run tests, you can access recorded diagnostics using the `DiagnosticRecord` field in the `Details` property on the `TestResult` object. For example, if your test results are stored in the variable `results`, then `result(2).Details.DiagnosticRecord` contains the recorded diagnostics for the second test in the suite.

The recorded diagnostics are `DiagnosticRecord` objects. To access particular types of test diagnostics for a 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.Verbose.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 34-95 and the `plugins` classes.

See Also

`TestRunner` | `TestSuite` | `matlab.unittest.constraints` | `plugins` | `selectors`

Related Examples

- “Add Plugin to Test Runner” on page 34-95

Run Tests in Editor

When you open a function-based or class-based test file in the MATLAB Editor, you have the option to run all tests in the file or to run the test at your cursor location. (This functionality is also available when you open a function-based test file in the Live Editor.) This example shows how to run tests while working in the Editor.

In the Editor, create a test in a file named `sampleTest.m`.

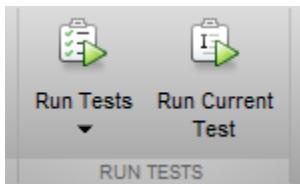
```
function tests = sampleTest
    tests = functiontests(localfunctions);
end

function testA(testCase)
    verifyEqual(testCase,5,5)
end

function testB(testCase)
    verifyGreaterThanOrEqual(testCase,42,13)
end

function testC(testCase)
    verifySubstring(testCase,'hello, world','llo')
end
```

When you save the test, the **Run** section in the **Editor** tab changes to **Run Tests**.



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 **Run Current Test**. 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]

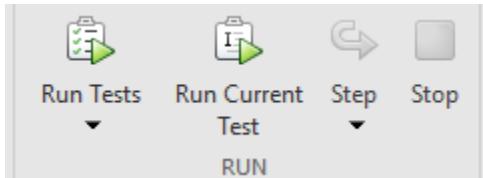
Totals:
1 Passed, 0 Failed, 0 Incomplete.
0.00099411 seconds testing time.
```

In addition to running tests, you can customize the test run using the test options under **Run Tests**. (Click the down arrow under **Run Tests** to access the full option list.) MATLAB uses test options whether you run all the tests in a file or just the test at your cursor location. When you select a test option, the selection persists for the duration of your current MATLAB session.

Test Option	Description
Clear Output Before Running Tests	Clears the Command Window before running tests.
Strict	Applies strict checks while running tests. For example, the framework generates a qualification failure if a test issues a warning. Tests that run with this option selected have the ' <code>Strict</code> ' option of <code>runtests</code> set to <code>true</code> .
Parallel	Runs tests in parallel. This option is only available if Parallel Computing Toolbox is installed. Tests that run with this option selected have the ' <code>UseParallel</code> ' option of <code>runtests</code> set to <code>true</code> .

Test Option	Description
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 runtests set to 0.</p>
Logging Level	<p>Displays diagnostics logged by the log 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 runtests set to 3.</p>

You can also run the tests of this example in the Live Editor by saving them in a file with a .mlx extension and using the **Run** section in the **Live Editor** tab.



See Also

[runtests](#)

Write Function-Based Unit Tests

In this section...

- “Create Test Function” on page 34-20
- “Run the Tests” on page 34-22
- “Analyze the Results” on page 34-23

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. The name of the main function corresponds to the name of your test file and should start or end with the word ‘test’, which is case-insensitive. If the file name does not start or end with the word ‘test’, the tests in the file might be ignored in certain cases. 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 testing framework automatically generates this object. For more information on creating test functions, see “Write Simple Test Case Using Functions” on page 34-24 and “Table of Verifications, Assertions, and Other Qualifications” on page 34-45. This is a typical example of skeletal local-test functions.

```
function testFunctionOne(testCase)
% Test specific code
end

function testFunctionTwo(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 testing 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 34-27.

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 testFunctionTwo(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

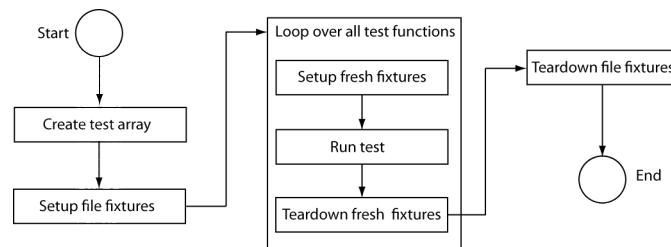
%% Optional fresh fixtures
function setup(testCase) % do not change function name
% open a figure, for example
end

function teardown(testCase) % do not change function name
% close figure, for example
end
```

Run the Tests

When you run function-based tests, the testing framework executes these tasks:

- 1 Create an array of tests specified by local test functions.
- 2 If the `setupOnce` function is specified, set up the pretest state of the system by running the function.
- 3 For each test, run the corresponding local test function. If the `setup` function is specified, run it before running the local test function. If the `teardown` function is specified, run it after running the local test function.
- 4 If the `teardownOnce` function is specified, return the pretest state of the system to the original state by running the function.



To run tests from the command prompt, use the `runtests` function 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 `runtests` and “Run Tests for Various Workflows” on page 34-91.

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 did not complete, and the time it took to run the test. For more information, see “Analyze Test Case Results” on page 34-125 and “Analyze Failed Test Results” on page 34-128.

See Also

`functiontests` | `localfunctions` | `runtests`

Related Examples

- “Write Simple Test Case Using Functions” on page 34-24
- “Write Test Using Setup and Teardown Functions” on page 34-27

Write Simple Test Case Using Functions

You can test your MATLAB® program by defining unit tests within a single file that contains a main function and local test functions. In a function-based test, each local function executes a portion of the software and qualifies the correctness of the produced result. For more information about function-based tests, see “Write Function-Based Unit Tests” on page 34-20.

This example shows how to write a function-based test to qualify the correctness of a function defined in a file in your current folder. The `quadraticSolver` function takes as inputs the coefficients of a quadratic polynomial and returns the roots of that polynomial. If the coefficients are specified as nonnumeric values, the function throws an error.

```
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 Tests

To test the `quadraticSolver` function, create the test file `quadraticSolverTest.m` in your current folder. Then, define a main function and two local functions in the file to test `quadraticSolver` against real and imaginary solutions. The name of the main and local functions must start or end with the word “test”, which is case-insensitive. Additionally, the name of the main function must correspond to the name of your test file.

Define Main Function

To run function-based unit tests, you must define a main function that collects all of the local test functions into a test array. Define the main function `quadraticSolverTest` in your test file. The main function calls `functiontests` to generate the test array `tests`. Pass `localfunctions` to `functiontests` to automatically generate a cell array of function handles to the local functions in your file.

```
function tests = quadraticSolverTest
tests = functiontests(localfunctions);
end
```

Define Local Test Functions

Add local functions to the test file to test the `quadraticSolver` function against real and imaginary solutions. The order of the tests within the file does not matter. Each local function must accept a single input `testCase`, which is a `matlab.unittest.FunctionTestCase` object. The testing framework automatically generates this object. The function uses the object to perform qualifications for testing values and responding to failures.

Define a local function `testRealSolution` to verify that `quadraticSolver` returns the correct real solutions for specific coefficients. For example, the equation $x^2 - 3x + 2 = 0$ has real solutions $x = 1$

and $x = 2$. The function calls `quadraticSolver` with the coefficients of this equation. Then, it uses the `verifyEqual` qualification function to compare the actual output `actSolution` to the expected output `expSolution`.

```
function tests = solverTest
tests = functiontests(localfunctions);
end

function testRealSolution(testCase)
actSolution = quadraticSolver(1,-3,2);
expSolution = [2 1];
verifyEqual(testCase,actSolution,expSolution)
end
```

Define a second local function `testImaginarySolution` to verify that `quadraticSolver` returns the correct imaginary solutions for specific coefficients. For example, the equation $x^2 + 2x + 10 = 0$ has imaginary solutions $x = -1 + 3i$ and $x = -1 - 3i$. Just like the previous function, this function calls `quadraticSolver` with the coefficients of this equation, and then uses the `verifyEqual` qualification function to compare the actual output `actSolution` to the expected output `expSolution`.

```
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 Test File

Use the `runtests` function to run the tests defined in the `quadraticSolverTest.m` file. In this example, both of the tests pass.

```
results = runtests('quadraticSolverTest.m')

Running quadraticSolverTest
.
.
Done quadraticSolverTest

_____

results =
1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details
```

```
Totals:  
 2 Passed, 0 Failed, 0 Incomplete.  
 0.89529 seconds testing time.
```

You also can run tests using the `run` function.

```
results = run(quadraticSolverTest)  
Running quadraticSolverTest  
. . .  
Done quadraticSolverTest  


---

  
results =  
 1x2 TestResult array with properties:  
  
  Name  
  Passed  
  Failed  
  Incomplete  
  Duration  
  Details  
  
Totals:  
 2 Passed, 0 Failed, 0 Incomplete.  
 0.017825 seconds testing time.
```

See Also

[functiontests](#) | [localfunctions](#) | [runtests](#)

More About

- “Write Function-Based Unit Tests” on page 34-20
- “Table of Verifications, Assertions, and Other Qualifications” on page 34-45
- “Analyze Test Case Results” on page 34-125
- “Create Simple Test Suites” on page 34-89

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.
  4.6534 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	De
{'axesPropertiesTest/testDefaultXLim' }	true	false	false	3.1401	{1x1}
{'axesPropertiesTest/surfaceColorTest'}	true	false	false	1.5133	{1x1}

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	De
{'axesPropertiesTest/surfaceColorTest'}	true	false	false	1.5133	{1x1}
{'axesPropertiesTest/testDefaultXLim' }	true	false	false	3.1401	{1x1}

See Also

`matlab.unittest.constraints` | `matlab.unittest.fixtures`

More About

- “Write Function-Based Unit Tests” on page 34-20
- “Table of Verifications, Assertions, and Other Qualifications” on page 34-45

Extend Function-Based Tests

In this section...

- “Fixtures for Setup and Teardown Code” on page 34-32
- “Test Logging and Verbosity” on page 34-33
- “Test Suite Creation” on page 34-33
- “Test Selection” on page 34-33
- “Test Running” on page 34-34
- “Programmatic Access of Test Diagnostics” on page 34-34
- “Test Runner Customization” on page 34-35

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 `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 `log` method. By default, the test runner reports diagnostics logged at verbosity level 1 (Terse). Use the `matlab.unittest.plugins.LoggingPlugin.withVerbosity` method to respond to messages of other verbosity levels. Construct a `TestRunner` object, add the `LoggingPlugin`, and run the suite with the `run` method. For more information on creating a test runner, see “Test Runner Customization” on page 34-35.

Test Suite Creation

Calling your function-based test returns a suite of `Test` objects. You also can use the `testsuite` function or the `matlab.unittest.TestSuite.fromFile` method. If you want a particular test and you know the test name, you can use `matlab.unittest.TestSuite.fromName`. If you want to create a suite from all tests in a particular folder, you can use `matlab.unittest.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 `matlab.unittest.TestSuite.fromFile`.
- Use a `selectors` instance and optional `constraints` instance.

Use these approaches in a suite generation method, such as `matlab.unittest.TestSuite.fromFile`, or create a suite and filter it using the `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 `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>run</code> with the suite
In a suite with a custom test runner	<code>run</code> . (See “Test Runner Customization” on page 34-35.)

For more information, see “Run Tests for Various Workflows” on page 34-91.

Programmatic Access of Test Diagnostics

In certain cases, the testing framework uses a `DiagnosticsRecordingPlugin` plugin to record diagnostics on test results. The framework uses the plugin by default if you do any of these:

- Run tests using the `runtests` function.
- Run tests using the `testrunner` function with no input.
- Run tests using the `run` method of the `TestSuite` or `TestCase` classes.
- Run performance tests using the `runperf` function.
- Run performance tests using the `run` method of the `TimeExperiment` class.

After you run tests, you can access recorded diagnostics using the `DiagnosticRecord` field in the `Details` property on the `TestResult` object. For example, if your test results are stored in the variable `results`, then `result(2).Details.DiagnosticRecord` contains the recorded diagnostics for the second test in the suite.

The recorded diagnostics are `DiagnosticRecord` objects. To access particular types of test diagnostics for a 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.Verbose.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 34-95 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 34-91
- “Add Plugin to Test Runner” on page 34-95

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
```

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.FatalAssert</code> <code>able</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 “Table of Verifications, Assertions, and Other Qualifications” on page 34-45.

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 34-39.

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 34-42.
- 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 34-66 and “Create Advanced Parameterized Test” on page 34-71.
- 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 34-51.

- Ability to create custom test fixtures. For more information see “Create Basic Custom Fixture” on page 34-54 and “Create Advanced Custom Fixture” on page 34-56.

See Also

Related Examples

- “Write Simple Test Case Using Classes” on page 34-39
- “Write Setup and Teardown Code Using Classes” on page 34-42
- “Create Simple Test Suites” on page 34-89
- “Run Tests for Various Workflows” on page 34-91
- “Analyze Test Case Results” on page 34-125
- “Analyze Failed Test Results” on page 34-128

Write Simple Test Case Using Classes

You can test your MATLAB® program by defining unit tests within a test class that inherits from the `matlab.unittest.TestCase` class. A unit test in a class-based test is a method that determines the correctness of a unit of software. It is defined within a `methods` block with the `Test` attribute and can use qualifications for testing values and responding to failures. For more information about class-based tests, see “Author Class-Based Unit Tests in MATLAB” on page 34-36.

This example shows how to write class-based unit tests to qualify the correctness of a function defined in a file in your current folder. The `quadraticSolver` function takes as inputs the coefficients of a quadratic polynomial and returns the roots of that polynomial. If the coefficients are specified as nonnumeric values, the function throws an error.

```
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

In a file in your current folder, create the `SolverTest` class by subclassing the `matlab.unittest.TestCase` class. This class provides a place for tests for the `quadraticSolver` function. Add three unit tests inside a `methods` block with the `Test` attribute. These test the `quadraticSolver` function against real solutions, imaginary solutions, and error conditions. Each `Test` method must accept a `TestCase` instance as an input. The order of the tests within the block does not matter.

First, create a `Test` method `realSolution` to verify that `quadraticSolver` returns the correct real solutions for specific coefficients. For example, the equation $x^2 - 3x + 2 = 0$ has real solutions $x = 1$ and $x = 2$. The method calls `quadraticSolver` with the coefficients of this equation. Then, it uses the `verifyEqual` method of `matlab.unittest.TestCase` to compare the actual output `actSolution` to the expected output `expSolution`.

```
classdef SolverTest < matlab.unittest.TestCase
    methods(Test)
        function realSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2 1];
            testCase.verifyEqual(actSolution,expSolution)
        end
    end
end
```

Create a second `Test` method `imaginarySolution` to verify that `quadraticSolver` returns the correct imaginary solutions for specific coefficients. For example, the equation $x^2 + 2x + 10 = 0$ has imaginary solutions $x = -1 + 3i$ and $x = -1 - 3i$. Just like the previous method, this method calls

`quadraticSolver` with the coefficients of this equation, and then uses the `verifyEqual` method to compare the actual output `actSolution` to the expected output `expSolution`.

```
classdef SolverTest < matlab.unittest.TestCase
    methods(Test)
        function realSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2 1];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function imaginarySolution(testCase)
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i -1-3i];
            testCase.verifyEqual(actSolution,expSolution)
        end
    end
end
```

Finally, add a Test method `nonnumericInput` to verify that `quadraticSolver` produces an error for nonnumeric coefficients. Use the `verifyError` method of `matlab.unittest.TestCase` to test that the function throws the error specified by '`quadraticSolver:InputMustBeNumeric`' when it is called with inputs 1, '-3', and 2.

```
classdef SolverTest < matlab.unittest.TestCase
    methods(Test)
        function realSolution(testCase)
            actSolution = quadraticSolver(1,-3,2);
            expSolution = [2 1];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function imaginarySolution(testCase)
            actSolution = quadraticSolver(1,2,10);
            expSolution = [-1+3i -1-3i];
            testCase.verifyEqual(actSolution,expSolution)
        end
        function nonnumericInput(testCase)
            testCase.verifyError(@()quadraticSolver(1,'-3',2), ...
                'quadraticSolver:InputMustBeNumeric')
        end
    end
end
```

Run Tests in SolverTest Class

To run all of the tests in the `SolverTest` class, create a `TestCase` object from the class and then call the `run` method on the object. In this example, all three tests pass.

```
testCase = SolverTest;
results = testCase.run

Running SolverTest
...
Done SolverTest



---


results =
1x3 TestResult array with properties:
```

```
Name
Passed
Failed
Incomplete
Duration
Details

Totals:
  3 Passed, 0 Failed, 0 Incomplete.
  1.2373 seconds testing time.
```

You also can run a single test specified by one of the `Test` methods. To run a specific `Test` method, pass the name of the method to `run`. For example, run the `realSolution` method.

```
result = run(testCase, 'realSolution')

Running SolverTest
.
Done SolverTest

_____

result =
  TestResult with properties:

    Name: 'SolverTest/realSolution'
    Passed: 1
    Failed: 0
    Incomplete: 0
    Duration: 0.0082
    Details: [1x1 struct]

Totals:
  1 Passed, 0 Failed, 0 Incomplete.
  0.0081829 seconds testing time.
```

See Also

`matlab.unittest.TestCase`

Related Examples

- “Author Class-Based Unit Tests in MATLAB” on page 34-36
- “Table of Verifications, Assertions, and Other Qualifications” on page 34-45
- “Analyze Test Case Results” on page 34-125
- “Create Simple Test Suites” on page 34-89

Write Setup and Teardown Code Using Classes

In this section...

["Test Fixtures" on page 34-42](#)

["Test Case with Method-Level Setup Code" on page 34-42](#)

["Test Case with Class-Level Setup Code" on page 34-43](#)

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)
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure)
        end
    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

```

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 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
```

See Also

[addTeardown](#) | [matlab.unittest.TestCase](#)

Related Examples

- “Author Class-Based Unit Tests in MATLAB” on page 34-36
- “Write Simple Test Case Using Classes” on page 34-39

Table of Verifications, Assertions, and Other Qualifications

There are four types of qualifications for testing values and responding to failures: verifications, assumptions, assertions, and fatal assertions.

- 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.	verifyTrue	assumeTrue	assertTrue	fatalAssertTrue
Value is false.	verifyFalse	assumeFalse	assertFalse	fatalAssertFalse
Value is equal to specified value.	verifyEqual	assumeEqual	assertEquals	fatalAssertEqual
Value is not equal to specified value.	verifyNotEqual	assumeNotEqual	assertNotEqual	fatalAssertNotEqual
Two values are handles to same instance.	verifySameHandle	assumeSameHandle	assertSameHandle	fatalAssertSameHandle
Value is not handle to specified instance.	verifyNotSameHandle	assumeNotSameHandle	assertNotSameHandle	fatalAssertNotSameHandle
Function returns true when evaluated.	verifyReturnsTrue	assumeReturnsTrue	assertReturnsTrue	fatalAssertReturnsTrue
Test produces unconditional failure.	verifyFail	assumeFail	assertFail	fatalAssertFail
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

Type of Test	Verification	Assumption	Assertion	Fatal Assertion
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
Function throws specified exception.	verifyError	assumeError	assertError	fatalAssertError
Function issues specified warning.	verifyWarning	assumeWarning	assertWarning	fatalAssertWarning
Function issues no warnings.	verifyWarningFree	assumeWarningFree	assertWarningFree	fatalAssertWarningFree

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
```

```
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 34-48
- “Select Tests Using TestSuite Methods” on page 34-49
- “Select Tests Using HasTag Selector” on page 34-49

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	Details
'ExampleTagTest/testE'	true	false	false	0.00039529	[1x1 struct]

```
'ExampleTagTest/testD'      true    false    false    0.00045658    [1x1 struct]
'ExampleTagClassTest/testH'  true    false    false    0.00043899    [1x1 struct]
```

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: [0x0 matlab.unittest.parameters.EmptyParameter]
SharedTestFixture: [0x0 matlab.unittest.fixtures.EmptyFixture]
Tags: {1x0 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
end
```

Create a Test for the BankAccount Class

Create a test file for the BankAccount class. Create the shared fixture by specifying the SharedTestFixture attribute for the TestCase and passing in a PathFixture.

BankAccountTest Class Definition File

```
classdef (SharedTestFixture={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

    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.assertTrue(callbackExecuted, ...
            'The listener callback should have fired')
    end
end
```

```
    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 the path.
```

```
_____  
Running BankAccountTest
```

```
.....  
Done BankAccountTest
```

```
_____  
Running DocPolynomTest
```

```
...  
Done DocPolynomTest
```

```
_____  
Tearing down PathFixture.
```

```
Description: Restores the path to its previous state.
```

The testing 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

`matlab.unittest.TestCase` | `matlab.unittest.fixtures` |
`matlab.unittest.fixtures.PathFixture`

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 how 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` class.

```
classdef FormatHexFixture < matlab.unittest.fixtures.Fixture
    properties (Access = private)
        OriginalFormat
    end
    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'
                      '3ff000000000000'
                      '4000000000000000'
                      '4008000000000000'];
            testCase.verifyEqual(actStr,expStr)
        end
    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 34-56
- “Write Tests Using Shared Fixtures” on page 34-51

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
```

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.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

        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
    methodsTestMethodSetup
        function mySetup(testCase)
            testCase.applyFixture(...
                UserNameEnvironmentVariableFixture('David'));
        end
    end

    methods (Test)
        function t1(~)
```

```
        fprintf(1, 'Current UserName: "%s"', getenv('UserName'))
    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 to "David"

Running testA
Current UserName: "David".
Done testA

Tearing down UserNameEnvironmentVariableFixture
Done tearing down UserNameEnvironmentVariableFixture: Restored the UserName environment variable

Setting up UserNameEnvironmentVariableFixture
Done setting up UserNameEnvironmentVariableFixture: Set the UserName environment variable to "Andy"

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 variable

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 does not 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.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.TearDownDescription = sprintf(...
        'Restored the UserName environment variable to "%s".', ...
        fixture.OriginalUser);
    setenv('UserName', fixture.OriginalUser)

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 `TearDownDescription`. 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 34-54
- “Write Tests Using Shared Fixtures” on page 34-51

Use Parameters in Class-Based Tests

Often, you need to run a series of tests that are different only in terms of test data. For example, you might want to test that a function produces the expected outputs for different inputs. In this case, the test logic is the same and the only difference between tests is the actual and expected values for each function call. With parameterized testing, you can implement code to iteratively run tests using different data values. When a method is parameterized, the testing framework automatically invokes the method for each parameter value. Therefore, you are not required to implement a separate method for each value.

The testing framework enables you to parameterize your test class at different levels. Additionally, when you call a test class method with multiple parameters, you can specify how the method should be invoked for different combinations of the parameters.

How to Write Parameterized Tests

Classes that derive from the `matlab.unittest.TestCase` class can implement test parameterization using framework-specific property and method attributes. To provide data as parameters in your class-based test, specify the data using a `properties` block with an appropriate parameterization attribute. Then, pass the parameterization property as an input argument to one or more methods.

For example, consider the `SampleTest` class. The class defines a parameterized test, because it specifies a property within a `properties` block with the `TestParameter` attribute. The property is passed to the `Test` method and is used to perform qualification.

```
classdef SampleTest < matlab.unittest.TestCase
    properties (TestParameter)
        Number = {1,2,'3',4,5};
    end
    methods(Test)
        function testDouble(testCase,Number)
            testCase.verifyClass(Number, 'double')
        end
    end
end
```

Because `Number` is a cell array with five elements, the test class results in a parameterized test suite with five elements.

```
suite = testsuite('SampleTest');
{suite.Name}

ans =
5x1 cell array

{'SampleTest/testDouble(Number=value1)'}
{'SampleTest/testDouble(Number=value2)'}
{'SampleTest/testDouble(Number=value3)'}
{'SampleTest/testDouble(Number=value4)'}
{'SampleTest/testDouble(Number=value5)'}
```

The value assigned to a parameterization property must be either a nonempty cell array or a scalar structure with at least one field. MATLAB uses the property value to specify parameter names and values in the test suite:

- If the property value is a cell array of character vectors, MATLAB generates parameter names from the values in the cell array. Otherwise, MATLAB specifies parameter names as `value1`, `value2`, ..., `valueN`.
- If the property value is a structure, structure fields represent the parameter names and structure values represent the parameter values. To include descriptive parameter names in the suite, define the parameters using a structure instead of a cell array.

How to Initialize Parameterization Properties

When you define a parameterization property, you must initialize the property so that MATLAB can generate parameter names and values. You can initialize the property at either test class load time or test suite creation time:

- **Class load time:** If the property value can be determined at the time MATLAB loads the test class definition, initialize the property using a default value. You can specify the default value in the `properties` block or using a local function in the `classdef` file. For more information about assigning values in a class definition, see “Evaluation of Expressions in Class Definitions”.

When you initialize a parameterization property at class load time, the parameters associated with the property remain fixed for different test runs. Each time you create a suite from the parameterized test class, the framework uses the same parameter names and values to run the tests. See “Create Basic Parameterized Test” on page 34-66 for an example using parameterization properties with default values.

- **Suite creation time:** If you cannot or do not want to determine the parameters at class load time, initialize the property at suite creation time using a static method with the `TestParameterDefinition` attribute. When you initialize a parameterization property with a `TestParameterDefinition` method, the parameters associated with the property can vary for different test runs. Each time you create a suite from the parameterized test class, the framework generates fresh parameter names and values to run the tests. For more information, see “Define Parameters at Suite Creation Time” on page 34-82.

Once you assign a value to a parameterization property, do not modify it. For example, when you initialize a parameterization property using a default value, you cannot use a `TestParameterDefinition` method to overwrite the default value.

A parameter might be used by several unit tests. Tests using the same parameter must run independently, without accidentally affecting one another. In addition, a running test must not affect subsequent reruns of the same test. To ensure test run independence, initialize your parameterization properties with value objects. Using handle objects (such as MATLAB graphics objects) as parameter values is not recommended. For more information about the behavior of value and handle objects, see “Comparison of Handle and Value Classes”.

If you need to test handle objects in your parameterized test, consider constructing them indirectly by using function handles as parameter values and invoking those function handles in tests. For example, write a parameterized test to test the default current point of figures created with the `figure` and `uifigure` functions.

```
classdef FigureTest < matlab.unittest.TestCase
    properties (TestParameter)
```

```

        FigureType = {@figure, @uifigure};
    end
methods(Test)
    function defaultCurrentPoint(testCase,FigureType)
        fig = FigureType();
        testCase.addTeardown(@close,fig)
        cp = fig.CurrentPoint;
        testCase.verifyEqual(cp,[0 0])
    end
end
end

```

Specify Parameterization Level

You can parameterize a test class at three levels: class setup, method setup, and test. Parameterizing at each level requires the parameterization properties to have a specific property attribute. For example, at the highest level, a `TestClassSetup` method can be parameterized using a property defined in a `properties` block with the `ClassSetupParameter` attribute. At the lowest level, a `Test` method can be parameterized using a property defined in a `properties` block with the `TestParameter` attribute.

This table shows different parameterization levels and the required method and property attributes for each level.

Parameterization Level	Parameterization Definition		Accessible Parameterization Properties
	Method Attribute	Property Attribute	
Class-setup level	<code>TestClassSetup</code>	<code>ClassSetupParameter</code>	<code>ClassSetupParameter</code>
Method-setup level	<code>TestMethodSetup</code>	<code>MethodSetupParameter</code>	<code>MethodSetupParameter</code> and <code>ClassSetupParameter</code>
Test level	<code>Test</code>	<code>TestParameter</code>	<code>TestParameter</code> , <code>MethodSetupParameter</code> , and <code>ClassSetupParameter</code>

A parameterized method can access parameterization properties depending on the level at which the method is defined:

- A parameterized `TestClassSetup` method can access parameterization properties only with the `ClassSetupParameter` attribute.
- A parameterized `TestMethodSetup` method can access parameterization properties only with the `MethodSetupParameter` or `ClassSetupParameter` attributes.
- A parameterized `Test` method can access any parameterization properties.

For an example of how to parameterize a test class at different levels, see “Create Advanced Parameterized Test” on page 34-71.

Specify How Parameters Are Combined

When you pass more than one parameterization property to a method, you can use the `ParameterCombination` method attribute to specify how parameters are combined. The testing framework invokes the method for the specified combinations.

This table shows different parameter combination strategies.

ParameterCombination Attribute	Method Invocation
'exhaustive' (default)	Methods are invoked for all combinations of parameter values. The testing 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. For example, if a method is provided with two parameterization properties and each property specifies three parameter values, then the framework invokes the method three times.
'pairwise'	Methods are invoked for every pair of parameter values at least one time. For example, if a method is provided with three parameterization properties, the testing framework guarantees that the method is invoked for every combination of parameter values specified by any two properties. Compared to 'exhaustive' combination, 'pairwise' combination typically results in fewer tests and therefore faster test execution. While the framework guarantees that tests are created for every pair of values at least one time, you should not rely on the suite size, ordering, or specific set of test suite elements.

You can combine parameters at class-setup, method-setup, and test levels. For example, use the combined method attributes `TestMethodSetup`, `ParameterCombination = 'sequential'` to specify sequential combination of the method-setup-level parameters defined in the `properties` block with the `MethodSetupParameter` attribute.

For examples of how to combine test parameters, see “Create Basic Parameterized Test” on page 34-66 and “Create Advanced Parameterized Test” on page 34-71.

Use External Parameters in Tests

When you create a parameterized test, you can redefine the parameters by injecting inputs into your class-based test. To provide data that is defined outside of the test file, create a `Parameter` instance and use the '`ExternalParameter`' option when creating your test suite. For more information, see “Use External Parameters in Parameterized Test” on page 34-78.

See Also

`matlab.unittest.TestCase` | `matlab.unittest.parameters` |
`matlab.unittest.parameters.Parameter`

More About

- “Create Basic Parameterized Test” on page 34-66
- “Create Advanced Parameterized Test” on page 34-71
- “Define Parameters at Suite Creation Time” on page 34-82
- “Use External Parameters in Parameterized Test” on page 34-78

Create Basic Parameterized Test

This example shows how to create a parameterized test to test the output of a function in terms of value, class, and size.

Create Function to Test

In your current 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(levels,classname)
if nargin == 1
    classname = 'single';
end

msize = 3^levels;
carpet = ones(msize,classname);

cutCarpet(1,1,msize,levels) % 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
```

Create TestCarpet Test Class

In a file in your current folder, create the `TestCarpet` class to test the `sierpinski` function. Define the properties used for parameterized testing in a `properties` block with the `TestParameter` attribute.

```
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
```

The `type` property contains the different data types you want to test. The `level` property contains the different fractal levels 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 parameter, `level` and `side` are defined as structures.

Define Test methods Block

In a `methods` block with the `Test` attribute, define three test methods:

- The `testRemainPixels` method tests the output of the `sierpinsk` 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 `sierpinsk` function with each combination of the `type` and `level` parameter values (that is, exhaustive parameter combination). The method results in nine test elements.
- The `testDefaultL10output` method does not use a `TestParameter` property and, therefore, is not parameterized. The method verifies that the level 1 matrix contains the expected values. Because the test method is not parameterized, it results in one test element.

```
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

    methods (Test)
        function testRemainPixels(testCase,level)
            expPixelCount = 8^level;
            actPixels = find(sierpinsk(level));
            testCase.verifyNumElements(actPixels,expPixelCount)
        end

        function testClass(testCase,type,level)
            testCase.verifyClass( ...
                sierpinsk(level,type),type)
        end

        function testDefaultL10output(testCase)
            exp = single([1 1 1; 1 0 1; 1 1 1]);
            testCase.verifyEqual(sierpinsk(1),exp)
        end
    end
end
```

Define Test methods Block with ParameterCombination Attribute

Define the `testNumel` method to ensure that the matrix returned by the `sierpinsk` function has the correct number of elements. Set the `ParameterCombination` attribute for the method to `'sequential'`. Because the `level` and `side` properties each specify three parameter values, the `testNumel` method is invoked three times — one time for each of the `'small'`, `'medium'`, and `'large'` values.

```
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

    methods (Test)
```

```
function testRemainPixels(testCase,level)
    expPixelCount = 8^level;
    actPixels = find(sierpinski(level));
    testCase.verifyNumElements(actPixels,expPixelCount)
end

function testClass(testCase,type,level)
    testCase.verifyClass( ...
        sierpinski(level,type),type)
end

function testDefaultL10output(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`. The suite has 16 test elements. MATLAB includes parameterization information in the names of the suite elements.

```
suite = matlab.unittest.TestSuite.fromFile('TestCarpet.m');
{suite.Name}

ans =

16×1 cell array

{'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/testDefaultL10output' }
```

Run the tests.

```
suite.run
```

```

Running TestCarpet
.....
Done TestCarpet
_____
ans =
1x16 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
    16 Passed, 0 Failed, 0 Incomplete.
    2.459 seconds testing time.

```

Run Tests with level Property Named 'small'

Use the `selectIf` method of `TestSuite` to select test elements that use a particular parameterization. Select all test elements that use the parameter name 'small' in the `level` parameterization property list. The filtered suite has five elements.

```

s1 = suite.selectIf('ParameterName','small');
{s1.Name}

ans =
5x1 cell array

{'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)'}

```

Run the filtered test suite.

```

s1.run;
Running TestCarpet
.....
Done TestCarpet
_____

```

Alternatively, you can create the same test suite directly using 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` |
`matlab.unittest.selectors.HasParameter`

Related Examples

- “Use Parameters in Class-Based Tests” on page 34-61
- “Create Advanced Parameterized Test” on page 34-71
- “Define Parameters at Suite Creation Time” on page 34-82
- “Use External Parameters in Parameterized Test” on page 34-78

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 random number generation.

Create `TestRand` Test Class

In a file in your current folder, create the `TestRand` class to test various aspects of random number generation. Define the properties used for parameterized testing.

```
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
end
```

In the `TestRand` class, each `properties` block corresponds to parameterization at a particular level. The 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 values.

Define Test Class and Test Method Setup Methods

Define the setup methods at the test class and test method levels. 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 `methodSetup` method seeds the generator.

```
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
end
```

Define Test Method with Sequential Parameter Combination

Define the `testSize` method in a `methods` block with the `Test` and `ParameterCombination = 'sequential'` attributes.

```
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
```

```
    end
end
```

The method tests the size of the output for each corresponding parameter value in `dim1`, `dim2`, and `dim3`. For a given `TestClassSetup` and `TestMethodSetup` parameterization, the framework calls the `testSize` method three times — one time for each of the '`small`', '`medium`', and '`large`' values. For example, to test with all of the '`medium`' values, the framework uses `testCase.verifySize(rand(2,3,4),[2 3 4])`.

Define Test Method with Pairwise Parameter Combination

Define the `testRepeatable` method in a `methods` block with the `Test` and `ParameterCombination = 'pairwise'` attributes.

```
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
```

The 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 with each pair of parameter values specified by `dim1`, `dim2`, and `dim3`. If the parameter combination were exhaustive, the framework would call the method $3^3 = 27$ times.

Define Test Method with Exhaustive Parameter Combination

Define the `testCase` method in a `methods` block with the `Test` attribute. Because the `ParameterCombination` attribute is not specified, the parameter combination is exhaustive by default.

```
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

```

The 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 $3 \times 3 \times 2 = 18$ times to ensure testing with each combination of `dim1`, `dim2`, and `type` parameter values.

Create Suite from Test Class

At the command prompt, create a suite from the `TestRand` 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.

```

For a given `TestClassSetup` and `TestMethodSetup` parameterization, the framework creates $3 + 10 + 18 = 31$ test elements. These 31 elements are called for each `TestMethodSetup` parameterization (resulting in $3 \times 31 = 93$ test elements for each `TestClassSetup` parameterization). There are three `TestClassSetup` parameterizations; therefore, the suite has a total of $3 \times 93 = 279$ test elements.

Query the name of the first test element.

```

suite(1).Name

ans =
'TestRand[generator=twister]/[seed=value1]testClass(dim1=small,dim2=small,type=single)'

```

The name of the first element is composed of these parts:

- Test class: `TestRand`

- Class setup property and parameter name: [generator=twister]
- Method setup property and parameter name: [seed=value1]
- Test method name: `testClass`
- Test method property and parameter names: (`dim1=small, dim2=small, type=single`)

The parameter name for the `seed` property is not particularly meaningful (`value1`). The testing framework provided this name because the `seed` property is specified as a cell array. For a more meaningful name, define the `seed` property as a structure with descriptive field names.

Run Suite Created Using Selector

At the command prompt, create a selector to select test elements that test the 'twister' generator for 'single' precision. Create a suite by omitting test elements that use properties with the 'large' parameter 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 =
1x12 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 from the `TestRand` class, you can construct the same filtered suite using the `selectIf` method.

```
suite = matlab.unittest.TestSuite.fromClass(?TestRand);
suite2 = selectIf(suite,s);
```

Run the filtered test suite.

```
suite2.run;

Running TestRand
.....
Done TestRand
```

Run Suite from Method Using Selector

Create a selector that omits test elements that use properties with the 'large' or 'medium' parameter names. 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 of the test elements from the `TestRand` class that use the `'double'` parameter name.

```

runtests('TestRand','ParameterName','double');

Running TestRand
.....
Done TestRand

```

See Also

`matlab.unittest.TestCase` | `matlab.unittest.TestSuite` |
`matlab.unittest.selectors.HasParameter`

Related Examples

- “Use Parameters in Class-Based Tests” on page 34-61
- “Create Basic Parameterized Test” on page 34-66
- “Define Parameters at Suite Creation Time” on page 34-82
- “Use External Parameters in Parameterized Test” on page 34-78

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
        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.AreEqual
            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	Duration
{'TestClean/classCheck(Data=clean)' }	true	false	false	0.19182
{'TestClean/classCheck(Data=needsCleaning)' }	true	false	false	0.0037433
{'TestClean/sortCheck(Data=clean)' }	true	false	false	0.0030112
{'TestClean/sortCheck(Data=needsCleaning)' }	true	false	false	0.0023779
{'TestClean/finiteCheck(Data=clean)' }	true	false	false	0.044825
{'TestClean/finiteCheck(Data=needsCleaning)' }	true	false	false	0.0031761
{'TestClean/noZeroCheck(Data=clean)' }	true	false	false	0.73055
{'TestClean/noZeroCheck(Data=needsCleaning)' }	true	false	false	0.0070594

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 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

- “Use Parameters in Class-Based Tests” on page 34-61
- “Create Basic Parameterized Test” on page 34-66
- “Create Advanced Parameterized Test” on page 34-71
- “Define Parameters at Suite Creation Time” on page 34-82

Define Parameters at Suite Creation Time

Parameterized tests let you run the same test procedure repeatedly, using different data values each time. In a parameterized test, these data values are called *parameters* and are represented by parameterization properties of the test class. MATLAB® uses parameterization properties to generate the parameter names and values for each test run.

In most cases, MATLAB can determine the value of a parameterization property when it loads the test class definition. Therefore, you can initialize the property using a default value. When you initialize a parameterization property with a default value, the parameters associated with the property remain fixed for different test runs. Each time you create a suite from the parameterized test class, the testing framework uses the same parameter names and values to run the tests.

In some cases, MATLAB cannot determine the value of a parameterization property when it loads the test class definition. For example, sometimes a parameterization property depends on another property defined at a higher parameterization level. Or you might not want the parameters to be determined at class load time. For instance, if parameters represent files in a folder, you might want to refresh the parameters each time you create a suite to test the files. When you cannot or do not want to initialize a parameterization property at class load time, initialize it at suite creation time using a static method with the `TestParameterDefinition` attribute. When you initialize a parameterization property using a `TestParameterDefinition` method, the parameters associated with the property can vary for different test runs. In other words, each time you create a test suite from the parameterized test class, the framework generates fresh parameter names and values to run the tests.

This example shows how to use parameterization properties with default and nondefault values to verify that the public properties of a group of classes in your current folder are nonempty. In it, you define a parameterized test class named `PropertiesTest` in the `test` subfolder of your current folder. You define three classes to test, named `ClassA`, `ClassB`, and `ClassC`, in the `source` subfolder of your current folder. For a summary of these three classes, see [Classes in source Subfolder on page 34-0](#).

Create `PropertiesTest` Class

To test the public properties of classes defined in the `source` subfolder, create the `PropertiesTest` class in the `test` subfolder. This class takes three specified classes, retrieves all properties of each class, and verifies that they are nonempty. To iterate over the classes to test, parameterize `PropertiesTest` at the class-setup level. To iterate over the properties of each class specified by a given class-setup-level parameterization, parameterize `PropertiesTest` at the test level.

Define the properties used for parameterized testing:

- List the classes for the framework to iterate over in a property named `ClassToTest`. Because this example assumes that the classes in the `source` subfolder are fixed and known at the time MATLAB loads the test class definition, initialize the property using a default value. In order to specify the class to test before running any `Test` methods, make `ClassToTest` a `ClassSetupParameter` property.
- Define a `TestParameter` property named `PropertyToTest` that you can use to iterate over the properties of whatever class the framework is currently testing. Because its value depends on the class being tested, do not assign it a default value. Instead, initialize it at suite creation time using a `TestParameterDefinition` method.
- To store the value of different properties on an instance of the class being tested, define a property named `ObjectToTest`.

```

classdef PropertiesTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        ClassToTest = {'ClassA', 'ClassB', 'ClassC'};
    end

    properties (TestParameter)
        PropertyToTest
    end

    properties
        ObjectToTest
    end
end

```

Define Method to Initialize Test-Level Parameterization Property

In the `PropertiesTest` class, `PropertyToTest` has a different value for each class being tested. Therefore, you cannot assign a default value to it. Instead, you must initialize it at suite creation time. To implement this requirement, add a `TestParameterDefinition` method named `initializeProperty`. Because a `TestParameterDefinition` method must be static, use the combined method attributes `TestParameterDefinition`, `Static` to define the method.

The `initializeProperty` method accepts the class-setup-level parameterization property as an input and uses it in a call to the `properties` function to return property names in a cell array of character vectors.

```

classdef PropertiesTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        ClassToTest = {'ClassA', 'ClassB', 'ClassC'};
    end

    properties (TestParameter)
        PropertyToTest
    end

    properties
        ObjectToTest
    end

    methods (TestParameterDefinition, Static)
        function PropertyToTest = initializeProperty(ClassToTest)
            PropertyToTest = properties(ClassToTest);
        end
    end
end

```

In the `initializeProperty` method, both the input argument `ClassToTest` and the output argument `PropertyToTest` are parameterization properties defined in the `PropertiesTest` class. Any time you define a `TestParameterDefinition` method, all inputs to the method match parameterization properties defined in the same class or one of its superclasses. Also, all outputs of the method must match parameterization properties defined in the same class.

In the `initializeProperty` method, the input argument `ClassToTest` is defined at the highest parameterization level. This puts it higher than the output argument `PropertyToTest`, which is defined at the lowest parameterization level. Any time you define a `TestParameterDefinition` method that accepts inputs, the inputs must be at a higher parameterization level relative to the

outputs of the method. For more information about parameterization levels, see “Use Parameters in Class-Based Tests” on page 34-61.

Define Test Class Setup Method

To test for nonempty property values, you must first create an object of the class being tested so that you can retrieve the property values. To implement this requirement, add the parameterized `classSetup` method to the `PropertiesTest` class. In order to have the object ready before running any `Test` methods, make `classSetup` a `TestClassSetup` method.

The `classSetup` method creates an instance of the class being tested and stores it in the `ObjectToTest` property. Tests can later retrieve the property values from `ObjectToTest`. In this example, the framework runs the tests by calling the `classSetup` method three times—one time for each of `ClassA`, `ClassB`, and `ClassC`.

```
classdef PropertiesTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        ClassToTest = {'ClassA', 'ClassB', 'ClassC'};
    end

    properties (TestParameter)
        PropertyToTest
    end

    properties
        ObjectToTest
    end

    methods (TestParameterDefinition, Static)
        function PropertyToTest = initializeProperty(ClassToTest)
            PropertyToTest = properties(ClassToTest);
        end
    end

    methods (TestClassSetup)
        function classSetup(testCase,ClassToTest)
            constructor = str2func(ClassToTest);
            testCase.ObjectToTest = constructor();
        end
    end
end
```

Test for Nonempty Property Values

To test that the properties on `ObjectToTest` are nonempty, add a `Test` method named `testProperty`. In order for the method to iterate over the properties of `ObjectToTest`, make the method parameterized, and pass it `PropertyToTest`.

During each test, the `testProperty` method retrieves the value of a property on `ObjectToTest`. Then, it uses a call to the `verifyNotEmpty` qualification method to verify that the value is not empty. For a given class-setup-level parameterization, the framework calls `testProperty` once for each property on the class being tested.

```
classdef PropertiesTest < matlab.unittest.TestCase
    properties (ClassSetupParameter)
        ClassToTest = {'ClassA', 'ClassB', 'ClassC'};
    end
```

```

properties (TestParameter)
    PropertyToTest
end

properties
    ObjectToTest
end

methods (TestParameterDefinition, Static)
    function PropertyToTest = initializeProperty(ClassToTest)
        PropertyToTest = properties(ClassToTest);
    end
end

methods (TestClassSetup)
    function classSetup(testCase,ClassToTest)
        constructor = str2func(ClassToTest);
        testCase.ObjectToTest = constructor();
    end
end

methods (Test)
    function testProperty(testCase,PropertyToTest)
        value = testCase.ObjectToTest.(PropertyToTest);
        testCase.verifyNotEmpty(value)
    end
end
end

```

Create Parameterized Test Suite and Run Tests

Now that the `PropertiesTest` class definition is complete, you can create a parameterized test suite and run the tests. To do this, make sure that the `source` and `test` subfolders are on the path.

```
addpath('source','test')
```

Create a suite from the `PropertiesTest` class.

```
suite = testsuite('PropertiesTest');
```

The test suite includes eight elements. Each element corresponds to a property defined within the `source` subfolder. Return the name of the first suite element.

```

suite(1).Name
ans =
'PropertiesTest[ClassToTest=ClassA]/testProperty(PropertyToTest=PropA1)'
```

The name of the first element is composed of these parts:

- `PropertiesTest` — Test class name
- `[ClassToTest=ClassA]` — Class-setup-level property and parameter name
- `testProperty` — Test method name
- `(PropertyToTest=PropA1)` — Test-level property and parameter name

Run the tests. Because two properties in the `source` subfolder are empty, two of the tests fail.

```

suite.run

Running PropertiesTest
..
=====
Verification failed in PropertiesTest[ClassToTest=ClassA]/testProperty(PropertyToTest=PropA3).
-----
Framework Diagnostic:
-----
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0 0].  

Actual Value:  

[]  

-----
Stack Information:  

-----
In C:\TEMP\Examples\matlab-ex41465327\test\PropertiesTest.m (PropertiesTest.testProperty) at
=====
....  

=====
Verification failed in PropertiesTest[ClassToTest=ClassC]/testProperty(PropertyToTest=PropC1).
-----
Framework Diagnostic:  

-----
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0 0].  

Actual Value:  

[]  

-----
Stack Information:  

-----
In C:\TEMP\Examples\matlab-ex41465327\test\PropertiesTest.m (PropertiesTest.testProperty) at
=====
..  

Done PropertiesTest
  

Failure Summary:  


| Name                                                                   | Failed | Incomplete |
|------------------------------------------------------------------------|--------|------------|
| PropertiesTest[ClassToTest=ClassA]/testProperty(PropertyToTest=PropA3) | X      |            |
| PropertiesTest[ClassToTest=ClassC]/testProperty(PropertyToTest=PropC1) | X      |            |

ans =
1x8 TestResult array with properties:  

Name
Passed
Failed
Incomplete
Duration
Details

```

```
Totals:
  6 Passed, 2 Failed (rerun), 0 Incomplete.
  2.9493 seconds testing time.
```

Run Tests for Specific Class

Run only the tests for `ClassB`. To do this, use the `selectIf` method of the `matlab.unittest.TestSuite` class to select test suite elements that use a particular parameterization. The resulting test suite is a *filtered suite* and has only three elements.

```
suite2 = suite.selectIf('ParameterName', 'PropB*');
{suite2.Name}

ans = 3x1 cell
  {'PropertiesTest[ClassToTest=ClassB]/testProperty(PropertyToTest=PropB1)'}
  {'PropertiesTest[ClassToTest=ClassB]/testProperty(PropertyToTest=PropB2)'}
  {'PropertiesTest[ClassToTest=ClassB]/testProperty(PropertyToTest=PropB3)'}
```

Run the filtered suite.

```
suite2.run;

Running PropertiesTest
...
Done PropertiesTest
```

Alternatively, you can run the same tests by creating a selector that filters the test suite by parameterization.

```
import matlab.unittest.selectors.HasParameter
import matlab.unittest.constraints.StartsWithSubstring
suite3 = matlab.unittest.TestSuite.fromClass(?PropertiesTest, ...
    HasParameter('Name',StartsWithSubstring('PropB')));
suite3.run;

Running PropertiesTest
...
Done PropertiesTest
```

Classes in source Subfolder

This section provides the contents of the classes in the `source` subfolder.

`ClassA` has three properties. Two of its properties have nonempty values.

```
classdef ClassA
properties
    PropA1 = 1;
    PropA2 = 2;
    PropA3
end
end
```

`ClassB` has three properties. All of its properties have nonempty values.

```
classdef ClassB
  properties
    PropB1 = 1;
    PropB2 = 2;
    PropB3 = 'a';
  end
end
```

ClassC has two properties. One of its properties has a nonempty value.

```
classdef ClassC
  properties
    PropC1
    PropC2 = [1 2 3];
  end
end
```

See Also

[matlab.unittest.TestCase](#) | [matlab.unittest.TestSuite](#) |
[matlab.unittest.selectors.HasParameter](#)

Related Examples

- “Use Parameters in Class-Based Tests” on page 34-61
- “Create Basic Parameterized Test” on page 34-66
- “Create Advanced Parameterized Test” on page 34-71
- “Use External Parameters in Parameterized Test” on page 34-78

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
```

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

`matlab.unittest.TestSuite`

Related Examples

- “Write Simple Test Case Using Classes” on page 34-39

Run Tests for Various Workflows

In this section...

- “Set Up Example Tests” on page 34-91
- “Run All Tests in Class or Function” on page 34-91
- “Run Single Test in Class or Function” on page 34-91
- “Run Test Suites by Name” on page 34-92
- “Run Test Suites from Test Array” on page 34-92
- “Run Tests with Customized Test Runner” on page 34-93

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 34-27.

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

`run (TestCase)` | `run (TestRunner)` | `run (TestSuite)` | `runtests`

More About

- “Run Tests in Editor” on page 34-17

Programmatically Access Test Diagnostics

In certain cases, the testing framework uses a `DiagnosticsRecordingPlugin` plugin to record diagnostics on test results. The framework uses the plugin by default if you do any of these:

- Run tests using the `runtests` function.
- Run tests using the `testrunner` function with no input.
- Run tests using the `run` method of the `TestSuite` or `TestCase` classes.
- Run performance tests using the `runperf` function.
- Run performance tests using the `run` method of the `TimeExperiment` class.

After you run tests, you can access recorded diagnostics using the `DiagnosticRecord` field in the `Details` property on the `TestResult` object. For example, if your test results are stored in the variable `results`, then `result(2).Details.DiagnosticRecord` contains the recorded diagnostics for the second test in the suite.

The recorded diagnostics are `DiagnosticRecord` objects. To access particular types of test diagnostics for a 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 34-95

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 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
        end
    end
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.assertTrue(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...

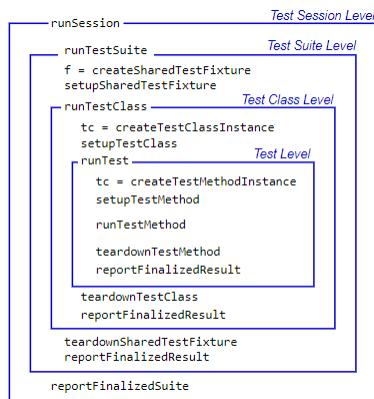
- ["Custom Plugins Overview" on page 34-97](#)
- ["Extending Test Session Level Plugin Methods" on page 34-97](#)
- ["Extending Test Suite Level Plugin Methods" on page 34-98](#)
- ["Extending Test Class Level Plugin Methods" on page 34-98](#)
- ["Extending Test Level Plugin Methods" on page 34-99](#)

Custom Plugins Overview

`TestRunnerPlugin` methods have four levels: test session, test suite, test class, and test. At each level, you implement methods to extend the running of tests. Additionally, you implement methods at the test suite, test class, and test levels to extend the creation, setup, and teardown of tests or test fixtures.

At the test suite, test class, and test levels, the `reportFinalizedResult` method enables the `TestRunner` to report finalized test results. A test result is finalized when no remaining test content can modify it. The `TestRunner` determines if it invokes the `reportFinalizedResult` method at each level. At the test session level, the `reportFinalizedSuite` method enables the `TestRunner` to report test results once the test suite is finalized.

The `TestRunner` runs different methods as shown in the figure.



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 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 Session Level Plugin Methods

The `TestRunnerPlugin` methods at the test session level extend the running and reporting of the test suite passed to the `TestRunner`. These methods fall within the scope of the `runSession` method.

The run method at this level, `runTestSuite`, extends the running of a portion of the entire `TestSuite` array that the testing framework passes to the `TestRunner`. The `reportFinalizedSuite` method extends the reporting of a test suite that has been finalized by `runTestSuite`.

Extending Test Suite Level Plugin Methods

The `TestRunnerPlugin` methods at the test suite level extend the creation, setup, running, and teardown of shared test fixtures. These methods fall within the scope of the `runTestSuite` method.

Type of Method	Test Level Falls Within Scope of <code>runTestSuite</code>
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.

Extending Test Class Level Plugin Methods

The `TestRunnerPlugin` methods at the test class level extend the creation, setup, running, 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` array that the `TestRunner` runs.

Type of Method	Test Level Falls Within Scope of <code>runTestClass</code>
creation method	<code>createTestClassInstance</code>
setup method	<code>setupTestClass</code>
run method	<code>runTest</code>
teardown method	<code>tearDownTestClass</code>

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 testing 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 testing 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 testing framework evaluates methods at the test class level within the scope of the `runTestClass` method. If the `TestClassSetup` code completes successfully, it invokes the `runTest` method one time for each element in the `TestSuite` array. Each `TestClassSetup` parameterization invokes the creation, setup, and teardown methods a single time.

Extending Test Level Plugin Methods

The `TestRunnerPlugin` methods at the test level extend the creation, setup, running, 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 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 testing framework passes each of these instances into the corresponding `Test` methods, and into any methods with the `TestMethodSetup` or `TestMethodTeardown` attribute.

The testing framework evaluates methods at the test level within the scope of the `runTest` method. Provided the framework completes all `TestMethodSetup` work, it invokes the plugin methods at this level a single time per `Test` element.

See Also

`addlistener` | `matlab.unittest.TestCase` | `matlab.unittest.TestRunner` |
`matlab.unittest.TestSuite` | `matlab.unittest.fixtures.Fixture` |
`matlab.unittest.plugins.OutputStream` | `matlab.unittest.plugins.Parallelizable` |
`matlab.unittest.plugins.TestRunnerPlugin`

Related Examples

- “Create Custom Plugin” on page 34-100
- “Run Tests in Parallel with Custom Plugin” on page 34-105
- “Plugin to Generate Custom Test Output Format” on page 34-122
- “Write Plugin to Save Diagnostic Details” on page 34-118

Create Custom Plugin

This example shows how to create a custom plugin that counts the number of passing and failing assertions when the `TestRunner` is running a test suite. The plugin prints a brief summary at the end of the testing. To extend the `TestRunner`, the plugin subclasses and overrides select methods of the `matlab.unittest.plugins.TestRunnerPlugin` class.

Create AssertionCountingPlugin Class

In a file in your current folder, create the custom plugin class `AssertionCountingPlugin`, which inherits from the `TestRunnerPlugin` class. For the complete code for `AssertionCountingPlugin`, see `AssertionCountingPlugin` Class Definition Summary on page 34-0 .

To keep track of the number of passing and failing assertions, define two read-only properties, `NumPassingAssertions` and `NumFailingAssertions`, within a `properties` block.

```
properties (SetAccess = private)
    NumPassingAssertions
    NumFailingAssertions
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 testing framework evaluates this method one time. It displays information about the total number of tests, initializes the properties used by the plugin to generate text output, and invokes the superclass method. After the framework completes evaluating the superclass method, the `runTestSuite` method displays the assertion count summary by calling the helper method `printAssertionSummary` (see Define Helper Methods on page 34-0).

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 used by the testing framework to create the test content. The test content includes `TestCase` instances for each `Test` element, class-level `TestCase` instances for the `TestClassSetup` and `TestClassTeardown` methods, and `Fixture` instances used when a `TestCase` class has the `SharedTestFixture` attribute.

Invoke the corresponding superclass method when you override the creation methods. The creation methods return the content that the testing framework creates for each of their respective contexts. When implementing one of these methods using the `incrementPassingAssertionsCount` and `incrementFailingAssertionsCount` helper methods on page 34-0 , add the listeners required by the plugin to the returned `Fixture` or `TestCase` instance.

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 method in a `methods` block with `protected` access. Like all plugin methods, the `runTest` method requires you to 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 Methods

In a `methods` block with `private` access, define three helper methods. These methods increment the number of passing or failing assertions, and print 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

The following code provides the complete contents of `AssertionCountingPlugin`.

```
classdef AssertionCountingPlugin < ...
    matlab.unittest.plugins.TestRunnerPlugin

    properties (SetAccess = private)
        NumPassingAssertions
        NumFailingAssertions
    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);
        end
    end
```

```

        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

    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 current folder, create a file named `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 `TestRunner` instance with no plugins. This code creates a silent runner and gives you 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 34-97
- “Run Tests in Parallel with Custom Plugin” on page 34-105
- “Write Plugin to Save Diagnostic Details” on page 34-118

Run Tests in Parallel with Custom Plugin

This example shows how to create a custom plugin that supports running tests in parallel. The custom plugin counts the number of passing and failing assertions for a test suite. To extend the `TestRunner`, the plugin overrides select methods of the `matlab.unittest.plugins.TestRunnerPlugin` class. Additionally, to support running tests in parallel, the plugin subclasses the `matlab.unittest.plugins.Parallelizable` interface. To run tests in parallel, you need Parallel Computing Toolbox.

Create Plugin Class

In a file in your current folder, create the parallelizable plugin class `AssertionCountingPlugin`, which inherits from both the `TestRunnerPlugin` and `Parallelizable` classes. For the complete code for `AssertionCountingPlugin`, see [Plugin Class Definition Summary](#) on page 34-0 .

To keep track of the number of passing and failing assertions, define four read-only properties within a `properties` block. Each MATLAB worker on the current parallel pool uses `NumPassingAssertions` and `NumFailingAssertions` to track the number of passing and failing assertions when running a portion of the `TestSuite` array. The MATLAB client uses `FinalizedNumPassingAssertions` and `FinalizedNumFailingAssertions` to aggregate the results from different workers and to report the total number of passing and failing assertions at the end of the test session.

```
properties (SetAccess = private)
    NumPassingAssertions
    NumFailingAssertions
    FinalizedNumPassingAssertions
    FinalizedNumFailingAssertions
end
```

Extend Running of Test Session

To extend the running of the entire `TestSuite` array, override the `runSession` method of `TestRunnerPlugin` in a `methods` block with `protected` access. The testing framework evaluates this method one time on the client.

```
methods (Access = protected)
    function runSession(plugin, pluginData)
        suiteSize = numel(pluginData.TestSuite);
        fprintf('## Running a total of %d tests\n\n', suiteSize);
        plugin.FinalizedNumPassingAssertions = 0;
        plugin.FinalizedNumFailingAssertions = 0;

        runSession@matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);

        fprintf('## Done running tests\n')
        plugin.printAssertionSummary()
    end
end
```

`runSession` displays information about the total number of `Test` elements, initializes the properties used by the plugin to generate text output, and invokes the superclass method to trigger the entire test run. After the framework completes evaluating the superclass method, `runSession` displays the assertion count summary by calling the helper method `printAssertionSummary` (see [Define Helper Methods](#) on page 34-0).

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 used by the testing framework to create the test content. The test content includes `TestCase` instances for each `Test` element, class-level `TestCase` instances for the `TestClassSetup` and `TestClassTeardown` method blocks, and `Fixture` instances used when a `TestCase` class has the `SharedTestFixture` attribute.

Invoke the corresponding superclass method when you override the creation methods. The creation methods return the content that the testing framework creates for each of their respective contexts. When implementing one of these methods using the `incrementPassingAssertionsCount` and `incrementFailingAssertionsCount` helper methods on page 34-0 , add the listeners required by the plugin to the returned `Fixture` or `TestCase` instance.

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 Test Suite Portion

The testing framework divides the entire `TestSuite` array into different groups and assigns them to workers for processing. Each worker can run one or more test suite portions. To customize the behavior of workers, override the `runTestSuite` method of `TestRunnerPlugin` in a `methods` block with `protected` access.

Extend the `TestRunner` to display the identifier of each test group that a worker runs along with the number of `Test` elements within the group. Additionally, store the number of passing and failing

assertions in a buffer so that the client can retrieve these values to produce the finalized results. Like all plugin methods, the `runTestSuite` method requires you to invoke the corresponding superclass method at an appropriate point. In this case, invoke the superclass method after initializing the properties and before storing the worker data. The testing framework evaluates `runTestSuite` on the workers as many times as the number of test suite portions.

```
methods (Access = protected)
    function runTestSuite(plugin, pluginData)
        suiteSize = numel(pluginData.TestSuite);
        groupNumber = pluginData.Group;
        fprintf('### Running a total of %d tests in group %d\n', ...
            suiteSize, groupNumber);
        plugin.NumPassingAssertions = 0;
        plugin.NumFailingAssertions = 0;

        runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(... ...
            plugin, pluginData);

        assertionStruct = struct('Passing', plugin.NumPassingAssertions, ...
            'Failing', plugin.NumFailingAssertions);
        plugin.storeIn(pluginData.CommunicationBuffer, assertionStruct);
    end
end
```

To store test-specific data, the implementation of `runTestSuite` contains a call to the `storeIn` method of the `Parallelizable` interface. Use `storeIn` along with `retrieveFrom` when workers must report to the client. In this example, after returning from the superclass method, `NumPassingAssertions` and `NumFailingAssertions` contain the number of passing and failing assertions corresponding to a group of tests. Because `storeIn` accepts the worker data as only one input argument, `assertionStruct` groups the assertion counts using two fields.

Extend Reporting of Finalized Test Suite Portion

Extend `reportFinalizedSuite` to aggregate the assertion counts by retrieving test data for each finalized test suite portion. To retrieve the stored `assertionStruct` for a test suite portion, invoke the `retrieveFrom` method within the scope of `reportFinalizedSuite`. Add the field values to the corresponding class properties, and invoke the superclass method. The testing framework evaluates this method on the client as many times as the number of test suite portions.

```
methods (Access = protected)
    function reportFinalizedSuite(plugin, pluginData)
        assertionStruct = plugin.retrieveFrom(pluginData.CommunicationBuffer);
        plugin.FinalizedNumPassingAssertions = ...
            plugin.FinalizedNumPassingAssertions + assertionStruct.Passing;
        plugin.FinalizedNumFailingAssertions = ...
            plugin.FinalizedNumFailingAssertions + assertionStruct.Failing;

        reportFinalizedSuite@matlab.unittest.plugins.TestRunnerPlugin(... ...
            plugin, pluginData);
    end
end
```

Define Helper Methods

In a `methods` block with `private` access, define three helper methods. These methods increment the number of passing or failing assertions within each running test suite portion, and print 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.FinalizedNumPassingAssertions + ...
            plugin.FinalizedNumFailingAssertions)
        fprintf('\t%d Passed, %d Failed\n', plugin.FinalizedNumPassingAssertions, ...
            plugin.FinalizedNumFailingAssertions)
    end
end
```

Plugin Class Definition Summary

The following code provides the complete contents of `AssertionCountingPlugin`.

```
classdef AssertionCountingPlugin < ...
    matlab.unittest.plugins.TestRunnerPlugin & ...
    matlab.unittest.plugins.Parallelizable

    properties (SetAccess = private)
        NumPassingAssertions
        NumFailingAssertions
        FinalizedNumPassingAssertions
        FinalizedNumFailingAssertions
    end

    methods (Access = protected)
        function runSession(plugin, pluginData)
            suiteSize = numel(pluginData.TestSuite);
            fprintf('## Running a total of %d tests\n\n', suiteSize);
            plugin.FinalizedNumPassingAssertions = 0;
            plugin.FinalizedNumFailingAssertions = 0;

            runSession@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

function runTestSuite(plugin, pluginData)
suiteSize = numel(pluginData.TestSuite);
groupNumber = pluginData.Group;
fprintf('### Running a total of %d tests in group %d\n', ...
    suiteSize, groupNumber);
plugin.NumPassingAssertions = 0;
plugin.NumFailingAssertions = 0;

runTestSuite@matlab.unittest.plugins.TestRunnerPlugin(... ...
    plugin, pluginData);

assertionStruct = struct('Passing', plugin.NumPassingAssertions, ...
    'Failing', plugin.NumFailingAssertions);
plugin.storeIn(pluginData.CommunicationBuffer, assertionStruct);
end

function reportFinalizedSuite(plugin, pluginData)
assertionStruct = plugin.retrieveFrom(pluginData.CommunicationBuffer);
plugin.FinalizedNumPassingAssertions = ...
    plugin.FinalizedNumPassingAssertions + assertionStruct.Passing;
plugin.FinalizedNumFailingAssertions = ...
    plugin.FinalizedNumFailingAssertions + assertionStruct.Failing;

reportFinalizedSuite@matlab.unittest.plugins.TestRunnerPlugin(... ...
    plugin, pluginData);
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.FinalizedNumPassingAssertions + ...
            plugin.FinalizedNumFailingAssertions);
        fprintf('\t%d Passed, %d Failed\n', plugin.FinalizedNumPassingAssertions, ...
            plugin.FinalizedNumFailingAssertions);
    end
end
end

```

Create Example Test Class

In your current folder, create a file named `ExampleTest.m` containing the following parameterized test class. This class results in 300 `Test` elements, 100 of which are assertion tests that compare pseudorandom integers between 1 and 10.

```

classdef ExampleTest < matlab.unittest.TestCase

properties (TestParameter)
    num1 = repmat({@()randi(10)}, 1, 10);
    num2 = repmat({@()randi(10)}, 1, 10);
end

```

```
methods(Test)
    function testAssert(testCase, num1, num2)
        testCase.assertNotEqual(num1(), num2())
    end
    function testVerify(testCase, num1, num2)
        testCase.verifyNotEqual(num1(), num2())
    end
    function testAssume(testCase, num1, num2)
        testCase.assumeNotEqual(num1(), num2())
    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 TestRunner instance with no plugins. This code creates a silent runner and gives you control over the installed plugins.

```
runner = TestRunner.withNoPlugins;
```

Add AssertionCountingPlugin to the runner and run the tests in parallel. You can also run the same tests in serial mode if you invoke the run method on the runner.

```
runner.addPlugin(AssertionCountingPlugin)
result = runner.runInParallel(suite);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
## Running a total of 300 tests

Split tests into 18 groups and running them on 6 workers.
-----
Finished Group 6
-----
### Running a total of 18 tests in group 6

-----
Finished Group 1
-----
### Running a total of 20 tests in group 1

-----
Finished Group 2
-----
### Running a total of 20 tests in group 2

-----
Finished Group 3
-----
### Running a total of 19 tests in group 3
```

```
-----  
Finished Group 4  
-----  
### Running a total of 19 tests in group 4  
  
-----  
Finished Group 5  
-----  
### Running a total of 18 tests in group 5  
  
-----  
Finished Group 7  
-----  
### Running a total of 18 tests in group 7  
  
-----  
Finished Group 8  
-----  
### Running a total of 17 tests in group 8  
  
-----  
Finished Group 9  
-----  
### Running a total of 17 tests in group 9  
  
-----  
Finished Group 10  
-----  
### Running a total of 17 tests in group 10  
  
-----  
Finished Group 11  
-----  
### Running a total of 16 tests in group 11  
  
-----  
Finished Group 12  
-----  
### Running a total of 16 tests in group 12  
  
-----  
Finished Group 15  
-----  
### Running a total of 15 tests in group 15  
  
-----  
Finished Group 14  
-----  
### Running a total of 15 tests in group 14  
  
-----  
Finished Group 17  
-----  
### Running a total of 14 tests in group 17  
  
-----  
Finished Group 16
```

```
-----  
### Running a total of 14 tests in group 16  
  
-----  
Finished Group 13  
-----  
### Running a total of 15 tests in group 13  
  
-----  
Finished Group 18  
-----  
### Running a total of 12 tests in group 18  
  
## Done running tests  


---

Total Assertions: 100  
88 Passed, 12 Failed
```

See Also

[addlistener](#) | [matlab.unittest.TestCase](#) | [matlab.unittest.TestResult](#) |
[matlab.unittest.TestRunner](#) | [matlab.unittest.TestSuite](#) |
[matlab.unittest.fixtures.Fixture](#) | [matlab.unittest.plugins.Parallelizable](#) |
[matlab.unittest.plugins.TestRunnerPlugin](#) | [runInParallel](#)

Related Examples

- “Write Plugins to Extend TestRunner” on page 34-97
- “Create Custom Plugin” on page 34-100

Write Plugin to Add Data to Test Results

This example shows how to create a plugin that adds data to `TestResult` objects. The plugin appends the actual and expected values in an assertion to the `Details` property of the `TestResult` object. To extend the `TestRunner`, the plugin overrides select methods of the `matlab.unittest.plugins.TestRunnerPlugin` class.

Create Plugin Class

In a file in your current folder, create the custom plugin class `DetailsRecordingPlugin`, which inherits from the `TestRunnerPlugin` class. For the complete code for `DetailsRecordingPlugin`, see `DetailsRecordingPlugin` Class Definition Summary on page 34-0 .

To store the actual and expected values in `TestResult` objects, define two constant properties, `ActField` and `ExpField`, within a `properties` block. Set the value of `ActField` to the name of the field of the `Details` structure that contains the actual value. Set the value of `ExpField` to the name of the field that contains the expected value.

```
properties (Constant,Access = private)
    ActField = 'ActualValue';
    ExpField = 'ExpectedValue';
end
```

Add Fields to Details Property

To add new fields to the `Details` property of all `TestResult` objects belonging to the test session, override the `runSession` method of `TestRunnerPlugin` in a `methods` block with `protected` access. `runSession` adds two empty fields to the `Details` structure of `TestResult` objects and invokes the superclass method to trigger the entire test run.

```
methods (Access = protected)
    function runSession(plugin,pluginData)
        resultDetails = pluginData.ResultDetails;
        resultDetails.append(plugin.ActField,{})
        resultDetails.append(plugin.ExpField,{})
        runSession@matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);
    end
end
```

To add the fields, the implementation of `runSession` contains calls to the `append` method of the `matlab.unittest.plugins.plugindata.ResultDetails` class. Each call adds an empty field to the `Details` structure.

Extend Creation of Shared Test Fixtures and TestCase Instances

Add listeners for the `AssertionPassed` and `AssertionFailed` events by extending the methods used by the testing framework to create the test content. The test content includes `TestCase` instances for each `Test` element, class-level `TestCase` instances for the `TestClassSetup` and `TestClassTeardown` method blocks, and `Fixture` instances used when a `TestCase` class has the `SharedTestFixture` attribute.

Invoke the corresponding superclass method when you override the creation methods. The listeners that you add to the returned `Fixture` or `TestCase` instances cause the `reactToAssertion` helper method on page 34-0 to execute whenever an assertion is performed. To add assertion data to test results, pass the `result` modifier instance along with the assertion event listener data to the helper method.

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);
        resultDetails = pluginData.ResultDetails;
        fixture.addlistener('AssertionPassed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
        fixture.addlistener('AssertionFailed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
    end

    function testCase = createTestClassInstance(plugin, pluginData)
        testCase = createTestClassInstance@...
        matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
        resultDetails = pluginData.ResultDetails;
        testCase.addlistener('AssertionPassed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
        testCase.addlistener('AssertionFailed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
    end

    function testCase = createTestMethodInstance(plugin, pluginData)
        testCase = createTestMethodInstance@...
        matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
        resultDetails = pluginData.ResultDetails;
        testCase.addlistener('AssertionPassed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
        testCase.addlistener('AssertionFailed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
    end
end
```

Define Helper Method

In a `methods` block with `private` access, define the helper method `reactToAssertion`. This method uses the `QualificationEventData` instance to extract the actual and expected values in assertions based on the `IsEqualTo` constraint, converts the extracted values to cell arrays, and appends the cell arrays to the fields of the corresponding `TestResult` object.

```
methods (Access = private)
    function reactToAssertion(plugin, evd, resultDetails)
        if ~isa(evd.Constraint,'matlab.unittest.constraints.IsEqualTo')
            return
        end
        resultDetails.append(plugin.ActField,{evd.ActualValue})
        resultDetails.append(plugin.ExpField,{evd.Constraint.Expected})
    end
end
```

DetailsRecordingPlugin Class Definition Summary

This code provides the complete contents of `DetailsRecordingPlugin`.

```
classdef DetailsRecordingPlugin < matlab.unittest.plugins.TestRunnerPlugin
    properties (Constant, Access = private)
        ActField = 'ActualValue';
        ExpField = 'ExpectedValue';
```

```

end

methods (Access = protected)
    function runSession(plugin,pluginData)
        resultDetails = pluginData.ResultDetails;
        resultDetails.append(plugin.ActField, {});
        resultDetails.append(plugin.ExpField, {});
        runSession@matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);
    end

    function fixture = createSharedTestFixture(plugin, pluginData)
        fixture = createSharedTestFixture@...
            matlab.unittest.plugins.TestRunnerPlugin(plugin, pluginData);
        resultDetails = pluginData.ResultDetails;
        fixture.addlistener('AssertionPassed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
        fixture.addlistener('AssertionFailed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
    end

    function testCase = createTestClassInstance(plugin,pluginData)
        testCase = createTestClassInstance@...
            matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);
        resultDetails = pluginData.ResultDetails;
        testCase.addlistener('AssertionPassed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
        testCase.addlistener('AssertionFailed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
    end

    function testCase = createTestMethodInstance(plugin,pluginData)
        testCase = createTestMethodInstance@...
            matlab.unittest.plugins.TestRunnerPlugin(plugin,pluginData);
        resultDetails = pluginData.ResultDetails;
        testCase.addlistener('AssertionPassed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
        testCase.addlistener('AssertionFailed',...
            @(~,evd)plugin.reactToAssertion(evd,resultDetails));
    end
end

methods (Access = private)
    function reactToAssertion(plugin, evd, resultDetails)
        if ~isa(evd.Constraint, 'matlab.unittest.constraints.AreEqual')
            return
        end
        resultDetails.append(plugin.ActField, {evd.ActualValue})
        resultDetails.append(plugin.ExpField, {evd.Constraint.Expected})
    end
end
end

```

Create Example Test Class

In your current folder, create a file named `ExampleTest.m` containing the following parameterized test class. The class results in a test suite with 25 elements, each corresponding to an experiment performed using a different seed for the random number generator. In each experiment, the testing framework creates a 1-by-100 vector of normally distributed random numbers and asserts that the magnitude of the difference between the actual and expected sample means is within 0.1.

```

classdef ExampleTest < matlab.unittest.TestCase
    properties
        SampleSize = 100;
    end

    properties (TestParameter)
        seed = num2cell(randi(10^6,1,25));
    end

```

```
methods(Test)
    function testMean(testCase,seed)
        import matlab.unittest.constraints.AreEqual
        import matlab.unittest.constraints.AbsoluteTolerance
        rng(seed)
        testCase.assertThat(mean(randn(1,testCase.SampleSize)),...
            equalTo(0,'Within',AbsoluteTolerance(0.1)));
    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 `TestRunner` instance with no plugins. This code creates a silent runner and gives you control over the installed plugins.

```
runner = TestRunner.withNoPlugins;
```

Add `DetailsRecordingPlugin` to the runner and run the tests.

```
runner.addPlugin(DetailsRecordingPlugin)
result = runner.run(suite)
```

```
result =
```

```
1x25 TestResult array with properties:
```

```
Name
Passed
Failed
Incomplete
Duration
Details
```

```
Totals:
```

```
18 Passed, 7 Failed (rerun), 7 Incomplete.
0.12529 seconds testing time.
```

To retrieve more information about the behavior of random number generation, create a structure array from the `Details` structures of the test results.

```
details = [result.Details]

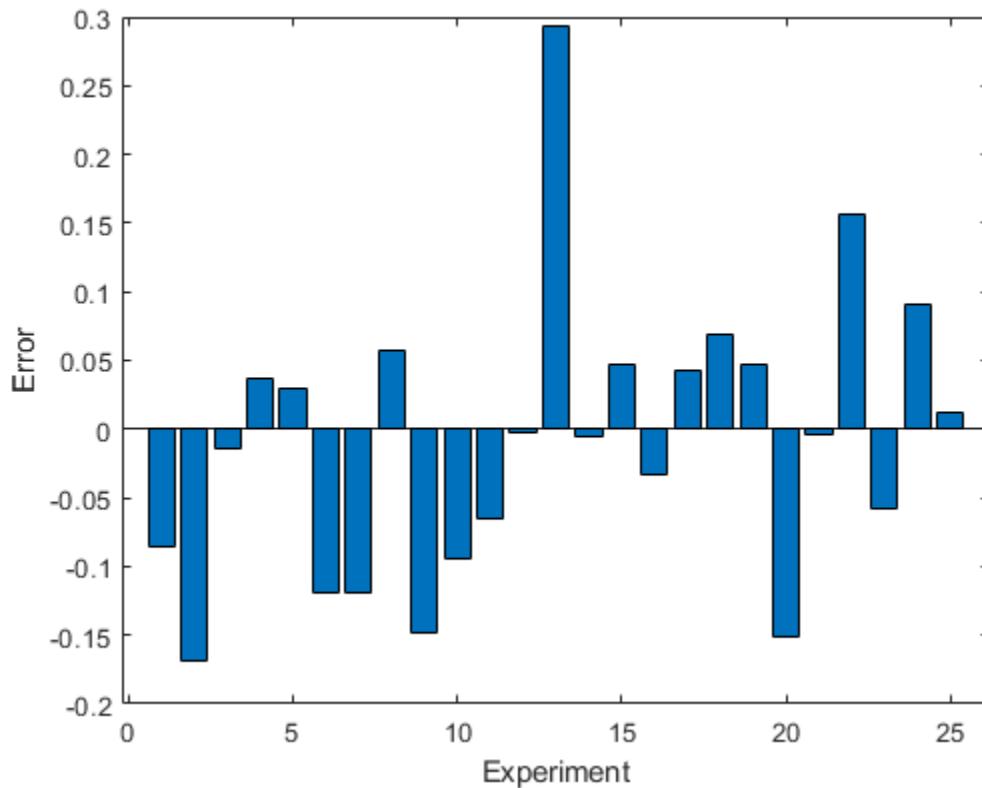
details =

1x25 struct array with fields:

    ActualValue
    ExpectedValue
```

Create an array containing the difference between the actual and expected values in each test and then display the error values in a bar graph. The seven bars with a length greater than 0.1 correspond to the failed tests.

```
errorInMean = cell2mat([details.ExpectedValue]) - cell2mat([details.ActualValue]);  
bar(errorInMean)  
xlabel('Experiment')  
ylabel('Error')
```



See Also

[addlistener](#) | [matlab.unittest.TestResult](#) | [matlab.unittest.TestRunner](#) |
[matlab.unittest.TestSuite](#) | [matlab.unittest.fixtures.Fixture](#) |
[matlab.unittest.plugins.TestRunnerPlugin](#) |
[matlab.unittest.plugins.plugindata.ResultDetails](#)

Related Examples

- “Write Plugins to Extend TestRunner” on page 34-97
- “Create Custom Plugin” on page 34-100

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

```

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.assertGreater(5,10)
        end
        function testTwo(testCase)
            wrongAnswer = 'wrong';
            testCase.verifyEmpty(wrongAnswer, 'Not Empty')
            testCase.verifyClass(wrongAnswer, 'double', 'Not double')
        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 =
```

```
5×6 table
```

Name	Type	TestDiagnostics
'ExampleTest/testOne'	'Assertion'	'TestDiagnostics not provided'
'ExampleTest/testTwo'	'Verification'	'Not Empty'
'ExampleTest/testTwo'	'Verification'	'Not double'
'ExampleTest/testThree'	'Assertion'	'Values not equal'
'ExampleTest/testFour'	'Fatal Assertion'	'TestDiagnostics not provided'

```
'assertGreaterThan failed.---> The value
'verifyEmpty failed.---> The value
'verifyClass failed.---> The value
'assertEqual failed.---> The values
'fatalAssertEqual failed.---> The v
```

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	-0.1666666666666667

```
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 34-97
- “Create Custom Plugin” on page 34-100
- “Plugin to Generate Custom Test Output Format” on page 34-122

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@...
                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)
        end
    end
end
```

```

        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.assertEqual(7*2,13,'Values not equal')
    end
    function testFour(testCase)
        testCase.verifyEqual(3+2,5);
    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 34-97
- “Write Plugin to Save Diagnostic Details” on page 34-118

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
methods(Test)
    function realSolution(testCase)
        actSolution = quadraticSolver(1,-3,2);
        expSolution = [2,1];
        testCase.verifyEqual(actSolution,expSolution)
    end
    function imaginarySolution(testCase)
        actSolution = quadraticSolver(1,2,10);
        expSolution = [-1+3i, -1-3i];
        testCase.verifyEqual(actSolution,expSolution)
    end
    function nonnumericInput(testCase)
        testCase.verifyError(@()quadraticSolver(1,'-3',2), ...
            'quadraticSolver:InputMustBeNumeric')
    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	1x3	7613	matlab.unittest.TestResult	

Display Information for One Test

To see the information for one value, type:

```
result(1)
```

```
ans =
  TestResult with properties:

    Name: 'SolverTest/realSolution'
    Passed: 1
    Failed: 0
    Incomplete: 0
    Duration: 0.0065
    Details: [1x1 struct]
```

Totals:

```
  1 Passed, 0 Failed, 0 Incomplete.
  0.0065241 seconds testing time.
```

Create Table of Test Results

To access functionality available to tables, create one from the `TestResult` object.

```
rt = table(result)
```

```
rt=3×6 table
```

Name	Passed	Failed	Incomplete	Duration	Details
{'SolverTest/realSolution' }	true	false	false	0.0065241	{1x1 struct}
{'SolverTest/imaginarySolution' }	true	false	false	0.0036673	{1x1 struct}
{'SolverTest/nonnumericInput' }	true	false	false	0.0074686	{1x1 struct}

Sort the test results by duration.

```
sortrows(rt, 'Duration')
```

```
ans=3×6 table
```

Name	Passed	Failed	Incomplete	Duration	Details
{'SolverTest/realSolution' }	true	false	false	0.0065241	{1x1 struct}
{'SolverTest/imaginarySolution' }	true	false	false	0.0036673	{1x1 struct}
{'SolverTest/nonnumericInput' }	true	false	false	0.0074686	{1x1 struct}

{'SolverTest/imaginarySolution'}	true	false	false	0.0036673	{1x1 struct}
{'SolverTest/realSolution'}	true	false	false	0.0065241	{1x1 struct}
{'SolverTest/nonnumericInput'}	true	false	false	0.0074686	{1x1 struct}

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 34-39

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 34-130

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 1×2 array that contains all the tests in `exampleTest.m`, and `ans` is a 1×1 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 34-128

Dynamically Filtered Tests

In this section...

["Test Methods" on page 34-133](#)

["Method Setup and Teardown Code" on page 34-135](#)

["Class Setup and Teardown Code" on page 34-136](#)

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.assertEqual(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.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
```

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.assertEqual(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.assertEqual(1,0)
        % remaining test code is not exercised
    end
end

methods(MethodSetup)
    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
```

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

`matlab.unittest.TestCase` | `matlab.unittest.TestResult` |
`matlab.unittest.qualifications.Assumable`

Create Custom Constraint

This example shows how to create a custom constraint that determines if a given value has the same size as an expected value.

In a file in your current folder, create a class named `HasSameSizeAs` that derives from the `matlab.unittest.constraints.Constraint` class. The class constructor accepts an expected value whose size is compared to the size of an actual value. The expected value is stored in the `ValueWithExpectedSize` property. The recommended practice is to make `Constraint` implementations immutable, so set the property `SetAccess` attribute to `immutable`.

```
classdef HasSameSizeAs < matlab.unittest.constraints.Constraint

    properties(SetAccess = immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
    end

end
```

In a `methods` block with `private` access, define a helper method `sizeMatchesExpected` that determines if the actual and expected values have the same size. This method is invoked by other constraint methods.

```
methods(Access = private)
    function bool = sizeMatchesExpected(constraint,actual)
        bool = isequal(size(actual),size(constraint.ValueWithExpectedSize));
    end
end
```

Classes that derive from the `matlab.unittest.constraints.Constraint` class must override the `satisfiedBy` method. This method must contain the comparison logic and return a logical value. Within a `methods` block, implement `satisfiedBy` by invoking the helper method. If the actual size and the expected size are equal, the method returns `true`.

```
methods
    function bool = satisfiedBy(constraint,actual)
        bool = constraint.sizeMatchesExpected(actual);
    end
end
```

Classes that derive from the `matlab.unittest.constraints.Constraint` class must also override 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` object.

```
methods
    function diag = getDiagnosticFor(constraint,actual)
        import matlab.unittest.diagnostics.StringDiagnostic
        if constraint.sizeMatchesExpected(actual)
            diag = StringDiagnostic('HasSameSizeAs passed.');
        end
    end
end
```

```
        else
            diag = StringDiagnostic(sprintf...
                'HasSameSizeAs failed.\nActual Size: [%s]\nExpectedSize: [%s]',...
                int2str(size(actual)),...
                int2str(size(constraint.ValueWithExpectedSize)) );
        end
    end
end
```

HasSameSizeAs Class Definition

This is the complete code for the HasSameSizeAs class.

```
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 = constraint.sizeMatchesExpected(actual);
        end

        function diag = getDiagnosticFor(constraint,actual)
            import matlab.unittest.diagnostics.StringDiagnostic
            if constraint.sizeMatchesExpected(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

    methods(Access = private)
        function bool = sizeMatchesExpected(constraint,actual)
            bool = isequal(size(actual),size(constraint.ValueWithExpectedSize));
        end
    end
end
```

Test for Expected Size

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)))
```

Verification passed.

Test a failing case.

```
testCase.verifyThat(zeros(5),HasSameSizeAs(ones(1,5)))
```

```
Verification failed.  
-----  
Framework Diagnostic:  
-----  
HasSameSizeAs failed.  
Actual Size: [5 5]  
ExpectedSize: [1 5]
```

See Also

[getDiagnosticFor](#) | [matlab.unittest.constraints.Constraint](#) |
[matlab.unittest.diagnostics.Diagnostic](#) |
[matlab.unittest.diagnostics.StringDiagnostic](#) | [satisfiedBy](#)

Related Examples

- “Create Custom Boolean Constraint” on page 34-142

Create Custom Boolean Constraint

This example shows how to create a custom Boolean constraint that determines if a given value has the same size as an expected value.

In a file in your current folder, create a class named `HasSameSizeAs` that derives from the `matlab.unittest.constraints.BooleanConstraint` class. The class constructor accepts an expected value whose size is compared to the size of an actual value. The expected value is stored in the `ValueWithExpectedSize` property. The recommended practice is to make `BooleanConstraint` implementations immutable, so set the property `SetAccess` attribute to `immutable`.

```
classdef HasSameSizeAs < matlab.unittest.constraints.BooleanConstraint

    properties(SetAccess = immutable)
        ValueWithExpectedSize
    end

    methods
        function constraint = HasSameSizeAs(value)
            constraint.ValueWithExpectedSize = value;
        end
    end
end
```

In a `methods` block with `private` access, define a helper method `sizeMatchesExpected` that determines if the actual and expected values have the same size. This method is invoked by other constraint methods.

```
methods(Access = private)
    function bool = sizeMatchesExpected(constraint,actual)
        bool = isequal(size(actual),size(constraint.ValueWithExpectedSize));
    end
end
```

The `matlab.unittest.constraints.BooleanConstraint` class is a subclass of the `matlab.unittest.constraints.Constraint` class. Therefore, classes that derive from the `BooleanConstraint` class must override the methods of the `Constraint` class. Within a `methods` block, override the `satisfiedBy` and `getDiagnosticFor` methods. The `satisfiedBy` implementation must contain the comparison logic and return a logical value. The `getDiagnosticFor` implementation must evaluate the actual value against the constraint and provide a `Diagnostic` object. In this example, `getDiagnosticFor` returns a `StringDiagnostic` object.

```
methods
    function bool = satisfiedBy(constraint,actual)
        bool = constraint.sizeMatchesExpected(actual);
    end

    function diag = getDiagnosticFor(constraint,actual)
        import matlab.unittest.diagnostics.StringDiagnostic
        if constraint.sizeMatchesExpected(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
```

Classes that derive from `BooleanConstraint` must implement the `getNegativeDiagnosticFor` method. This method must provide a `Diagnostic` object when the constraint is negated.

Override `getNegativeDiagnosticFor` in a `methods` block with `protected` access.

```
methods(Access = protected)
    function diag = getNegativeDiagnosticFor(constraint,actual)
        import matlab.unittest.diagnostics.StringDiagnostic
        if constraint.sizeMatchesExpected(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.');
        end
    end
end
```

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

This is the complete code for the `HasSameSizeAs` class.

```
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 = constraint.sizeMatchesExpected(actual);
        end

        function diag = getDiagnosticFor(constraint,actual)
            import matlab.unittest.diagnostics.StringDiagnostic

            if constraint.sizeMatchesExpected(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

    methods(Access = protected)
        function diag = getNegativeDiagnosticFor(constraint,actual)
            import matlab.unittest.diagnostics.StringDiagnostic
            if constraint.sizeMatchesExpected(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.');
            end
        end
    end
```

```
end

methods(Access = private)
    function bool = sizeMatchesExpected(constraint,actual)
        bool = isequal(size(actual),size(constraint.ValueWithExpectedSize));
    end
end

end
```

Test for Expected Size

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. The test passes because one of the `or` conditions, `HasLength(5)`, is true.

```
testCase.verifyThat(zeros(5),HasLength(5) | ~HasSameSizeAs(repmat(1,5)))
```

Verification passed.

Test a failing case. The test fails because one of the `and` conditions, `~HasSameSizeAs(repmat(1,5))`, is false.

```
testCase.verifyThat(zeros(5),HasLength(5) & ~HasSameSizeAs(repmat(1,5)))
```

Verification failed.

```
-----
Framework Diagnostic:
-----
AndConstraint failed.
--> + [First Condition]:
    | HasLength passed.

    Actual Value:
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   0   0
    |
    | Expected Length:
    |   5
--> AND
    + [Second Condition]:
        | Negated HasSameSizeAs failed.
        | Size [5 5] of Actual Value and Expected Value were the same but should not have been
```

See Also

`getDiagnosticFor` | `getNegativeDiagnosticFor` |
`matlab.unittest.constraints.BooleanConstraint` |
`matlab.unittest.constraints.Constraint` | `matlab.unittest.diagnostics.Diagnostic`
| `matlab.unittest.diagnostics.StringDiagnostic` | `satisfiedBy`

Related Examples

- “Create Custom Constraint” on page 34-139

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
```

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

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 `type`, 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 teardown 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.

Dismissing Alerts – In some cases, an app displays modal alert dialog boxes, which make it impossible to interact with app components. Accessing the figure behind a dialog box might require you to close the dialog box. To programmatically close an alert dialog box in the figure window, use the `dismissAlertDialog` method.

Gesture Support of UI Components

The gesture methods of `matlab.uitest.TestCase` support various UI components.

Component	Typical Creation Function	<code>matlab.uitest.TestCase</code> Gesture Method					
		press	choose	drag	type	hover	chooseContextMenu
Axes	axes	✓		✓		✓	✓
Button	uibutton	✓					✓

Button Group	uibuttongroup		✓					
Check Box	uicheckbox	✓	✓					✓
Date Picker	uidatepicker				✓			✓
Discrete Knob	uiknob		✓					✓
Drop Down	uidropdown		✓		✓			✓
Edit Field (Numeric, Text)	uieditfield				✓			✓
Image	uiimage	✓						✓
Knob	uiknob		✓	✓				✓
List Box	uilistbox		✓					✓
Menu	uimenu	✓						
Panel	uipanel	✓				✓		✓
Polar Axes	polaraxes	✓				✓		✓
Push Tool	uipushtool	✓						
Radio Button	uiradiobutton	✓	✓					✓
Slider	uislider		✓	✓				✓
Spinner	uispinner	✓			✓			✓
State Button	uibutton	✓	✓					✓
Switch (Rocker, Slider, Toggle)	uiswitch	✓	✓					✓
Tab	uitab		✓					
Tab Group	uitabgroup		✓					
Table	uitable		✓		✓			✓
Text Area	uitextarea				✓			✓
Toggle Button	uitogglebutton	✓	✓					✓
Toggle Tool	uitoggletool	✓	✓					

Tree Node	uitreenode		✓					✓
UI Axes	uiaxes	✓		✓			✓	✓
UI Figure	uifigure	✓					✓	✓

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.

At the command prompt, make the app accessible by adding the folder that includes the app to the MATLAB search path.

```
addpath(fullfile(matlabroot,'examples','matlab','main'))
```

To explore the properties of the 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.unittest.TestCase`.

```
classdef testConfigurePlotAppExample < matlab.unittest.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.unittest.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.unittest.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.
5.0835 seconds testing time.

See Also

`matlab.unittest.TestCase`

More About

- “Write Test for App” on page 34-155
- “Write Test That Uses App Testing and Mocking Frameworks” on page 34-159
- “App Building Components”

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.

At the command prompt, make the app accessible by adding the folder that includes the app to the MATLAB search path.

```
addpath(fullfile(matlabroot,'examples','matlab','main'))
```

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.unittest.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.unittest.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.DataTab;
            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.unittest.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.unittest.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 = testCase.App.DataTab.Children.Data;
        t.Gender = categorical(t.Gender);
        allMales = t(t.Gender == 'Male', :);
        maleDiastolicData = allMales.Diastolic';
        maleSystolicData = allMales.Systolic';

        % 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) ...

    function test_tab(testCase) ...

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 screenshot when the test fails, use a `ScreenshotDiagnostic` with the `onFailure` method.

```
classdef TestPatientsDisplay < matlab.unittest.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 = testCase.App.DataTab.Children.Data;
        t.Gender = categorical(t.Gender);
        allMales = t(t.Gender == 'Male',:);
        maleDiastolicData = allMales.Diastolic';
        maleSystolicData = allMales.Systolic';

        % 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.DataTab;
        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    126    137    138    137    118
        Columns 50 through 53
        141    129    124    134
-----
Additional Diagnostic:
-----
Screenshot captured to:
--> C:\Temp\83292efd-b703-46ef-8c41-00e20167321d\Screenshot_c025020f-281e-483c-8ca8-f1c857421fde.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 34-150
- “Write Test That Uses App Testing and Mocking Frameworks” on page 34-159

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.

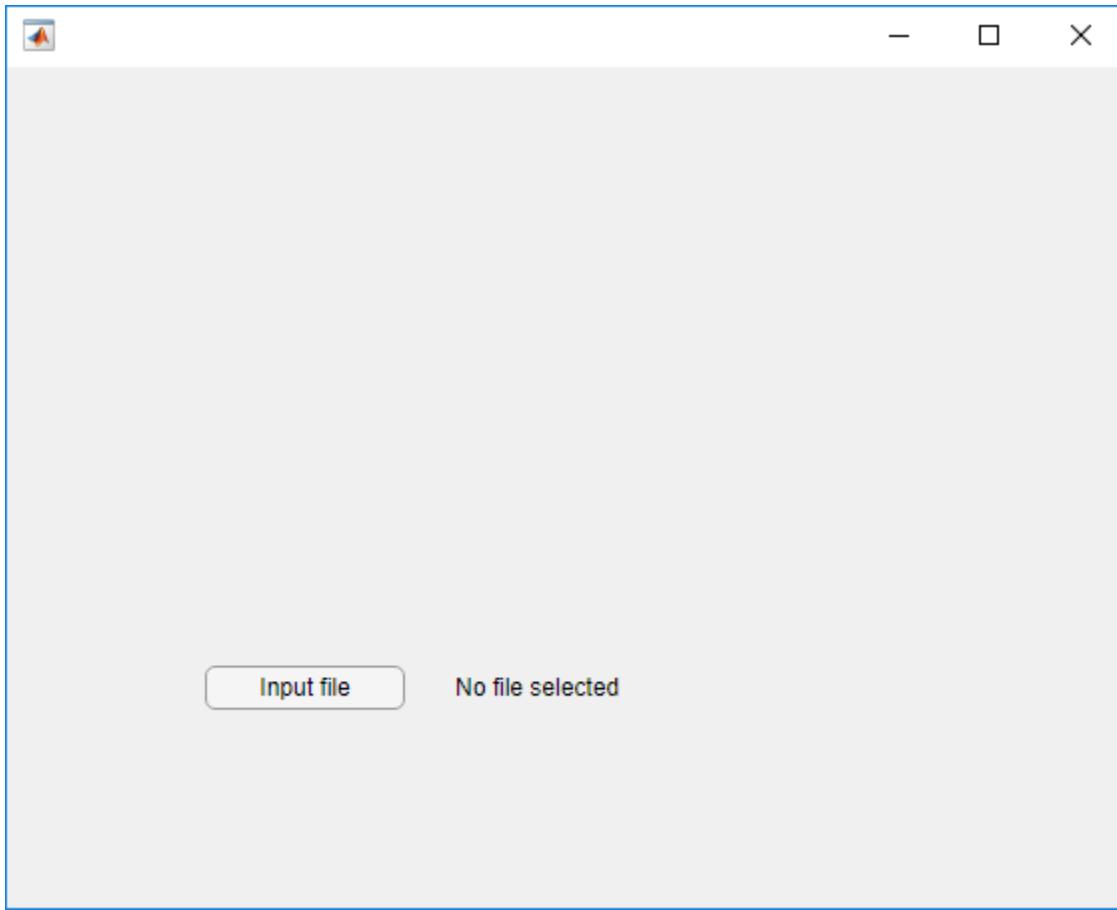
```
function app = launchApp
    f = uifigure;
    button = uibutton(f,'Text','Input file');
    button.ButtonPushedFcn = @(src,evt)pickFile;
    label = uilabel(f,'Text','No file selected');
    label.Position(1) = button.Position(1) + button.Position(3) + 25;
    label.Position(3) = 200;

    % 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.unittest.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

```

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.unittest.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.unittest.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);
    end
end
```

```
    tc.press(app.Button);
    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.unittest.TestCase`

More About

- “Overview of App Testing Framework” on page 34-150
- “Write Test for App” on page 34-155
- “Create Mock Object” on page 34-181

Overview of Performance Testing Framework

In this section...

- ["Determine Bounds of Measured Code" on page 34-164](#)
- ["Types of Time Experiments" on page 34-165](#)
- ["Write Performance Tests with Measurement Boundaries" on page 34-165](#)
- ["Run Performance Tests" on page 34-166](#)
- ["Understand Invalid Test Results" on page 34-166](#)

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
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

Type of Test	What Is Measured	What Is Excluded
Class-based deriving from <code>matlab.perftest.TestCase</code> and using the <code>keepMeasuring</code> method	Code inside each <code>keepMeasuring</code> -while loop in each method tagged with the <code>Test</code> attribute	<ul style="list-style-type: none"> Code outside of the <code>keepMeasuring</code>-while 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 `withFixedSize` 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.withFixedSize</code>
Number of samples	Between 4 and 256 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 testing framework accumulates and sums the measurements.

The performance testing 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 34-172.

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 256 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 `withFixedSize` 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 “Test Performance Using Classes” on page 34-172.

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 34-168

- “Test Performance Using Classes” on page 34-172

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 using `runperf`.

```
results = runperf('preallocationTest.m')

Running preallocationTest
.....
Done preallocationTest
```

```

results =
1x4 TimeResult array with properties:

  Name
  Valid
  Samples
  TestActivity

Totals:
  4 Valid, 0 Invalid.
  10.2561 seconds testing time.

```

The `results` variable is a 1-by-4 `TimeResult` 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 =
TimeResult with properties:

  Name: 'preallocationTest/testIndexingWithVariable'
  Valid: 1
  Samples: [4x7 table]
  TestActivity: [8x12 table]

Totals:
  1 Valid, 0 Invalid.
  1.2274 seconds testing time.

```

As indicated by the size of the `TestActivity` property, the performance testing framework collected 8 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

>> results(2).Samples

ans =
4x7 table

```

Name	MeasuredTime	Timestamp	Host
preallocationTest/testIndexingWithVariable	0.15271	24-Jun-2019 16:13:33	MY-HOST
preallocationTest/testIndexingWithVariable	0.15285	24-Jun-2019 16:13:33	MY-HOST
preallocationTest/testIndexingWithVariable	0.15266	24-Jun-2019 16:13:33	MY-HOST
preallocationTest/testIndexingWithVariable	0.15539	24-Jun-2019 16:13:34	MY-HOST

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 =
0.1534
```

The performance testing framework collected four sample measurements for the second test. The test took an average of 0.1534 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 =
4x3 table
```

Name	GroupCount	mean_MeasuredTime
preallocationTest/testOnes	4	0.041072
preallocationTest/testIndexingWithVariable	4	0.1534
preallocationTest/testIndexingOnLHS	4	0.04677
preallocationTest/testForLoop	4	1.0343

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 =
```

4×3 table

Name	GroupCount	mean_MeasuredTime
preallocationTest/testOnes	4	0.040484
preallocationTest/testIndexingWithVariable	4	0.15187
preallocationTest/testIndexingOnLHS	4	0.046224
preallocationTest/testForLoop	4	1.0262

See Also

[matlab.perftest.TimeExperiment](#) | [matlab.perftest.TimeResult](#) |

[matlab.unittest.measurement.DefaultMeasurementResult](#) | [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(TestMethod)
        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();

        testCase.verifyEqual(nbytes,length(textToWrite))
    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 TimeResult array with properties:

Name
Valid

```

```
Samples  
TestActivity  
  
Totals:  
  2 Valid, 0 Invalid.  
  4.1417 seconds testing time.
```

The `results` variable is a 1-by-2 `TimeResult` 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 =  
  
TimeResult with properties:  
  
  Name: 'fprintfTest/testPrintingToFile'  
  Valid: 1  
  Samples: [4x7 table]  
  TestActivity: [8x12 table]
```

```
Totals:  
  1 Valid, 0 Invalid.  
  2.7124 seconds testing time.
```

As indicated by the size of the `TestActivity` property, the performance testing framework collected 8 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 =
```

```
4x7 table
```

Name	MeasuredTime	Timestamp	Host	Platform
fprintfTest/testPrintingToFile	0.067729	24-Jun-2019 16:22:09	MY-HOSTNAME	windows
fprintfTest/testPrintingToFile	0.067513	24-Jun-2019 16:22:09	MY-HOSTNAME	windows
fprintfTest/testPrintingToFile	0.068737	24-Jun-2019 16:22:09	MY-HOSTNAME	windows
fprintfTest/testPrintingToFile	0.068576	24-Jun-2019 16:22:10	MY-HOSTNAME	windows

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 =
0.0681
```

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 =
```

2×3 table

Name	GroupCount	mean_MeasuredTime
fprintfTest/testPrintingToFile	4	0.068139
fprintfTest/testBytesToFile	9	0.071595

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 4 warm-up measurements and up to 16 sample 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',4,...  
    'MaxSamples',16,'RelativeMarginOfError',0.03,'ConfidenceLevel',0.97);
resultsTE = run(experiment,suite);
```

Running `fprintfTest`

```
..... Warning: Target Relative Margin of Error not met after running the MaxSample
```

.....

Done `fprintfTest`

In this example output, the performance testing framework is not able to meet the stricter statistical objectives with the specified 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 =
2x3 table

    Name        GroupCount    mean_MeasuredTime
    _____      _____
    fprintfTest/testPrintingToFile      16          0.069482
    fprintfTest/testBytesToFile        4           0.067902

```

Increase the maximum number of samples to 32 and rerun the time experiment.

```

experiment = TimeExperiment.limitingSamplingError('NumWarmups',4, ...
    'MaxSamples',32,'RelativeMarginOfError',0.03,'ConfidenceLevel',0.97);
resultsTE = run(experiment,suite);

Running fprintfTest
.....
Done fprintfTest

```

Compute the statistics for all the test elements.

```

fullTableTE = vertcat(resultsTE.Samples);
summaryStatsTE = varfun(@mean,fullTableTE, ...
    'InputVariables','MeasuredTime','GroupingVariables','Name')

summaryStatsTE =
2x3 table

    Name        GroupCount    mean_MeasuredTime
    _____      _____
    fprintfTest/testPrintingToFile      4           0.067228
    fprintfTest/testBytesToFile        4           0.067766

```

The testing framework achieves the statistical objectives for both tests with 4 samples.

Measure First-time Cost

Start a new MATLAB session. A new session ensures that MATLAB has not run the code contained in your tests.

Measure the first-time cost of your code by creating and running a fixed time experiment with zero warm-up measurements and one sample measurement.

Construct an explicit test suite. Since you are measuring the first-time cost of the function, run a single test. To run multiple tests, save the results and start a new MATLAB session between tests.

```
suite = testsuite('fprintfTest/testPrintingToFile');
```

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	Objective	Timestamp	Host
fprintfTest/testPrintingToFile	true	false	false	0.071754	sample	24-Jun-2019 16:31:27	MY-HOSTNAME

The performance testing framework collects one sample for each test.

See Also

`matlab.perftest.TestCase` | `matlab.perftest.TimeExperiment` |
`matlab.perftest.TimeResult` |
`matlab.unittest.measurement.DefaultMeasurementResult` | `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 `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
```

Run `PreallocationTest` as a performance test. Two 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/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/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

```

Rerun the tests. All the tests complete.

```
results = runperf('PreallocationTest');

Running PreallocationTest
.
.
.
Done PreallocationTest
```

View the results.

```
sampleSummary(results)
```

ans =

4x7 table

Name	SampleSize	Mean	StandardDeviation	Min	Median
PreallocationTest/testOnes	4	3.0804e-05	1.8337e-07	3.0577e-05	3.0843e-05
PreallocationTest/testIndexingWithVariable	4	0.00044536	1.7788e-05	0.00042912	0.000443
PreallocationTest/testIndexingOnLHS	4	5.6352e-05	1.8863e-06	5.5108e-05	5.5598e-05
PreallocationTest/testForLoop	4	0.0097656	0.00018202	0.0096181	0.0097000

See Also

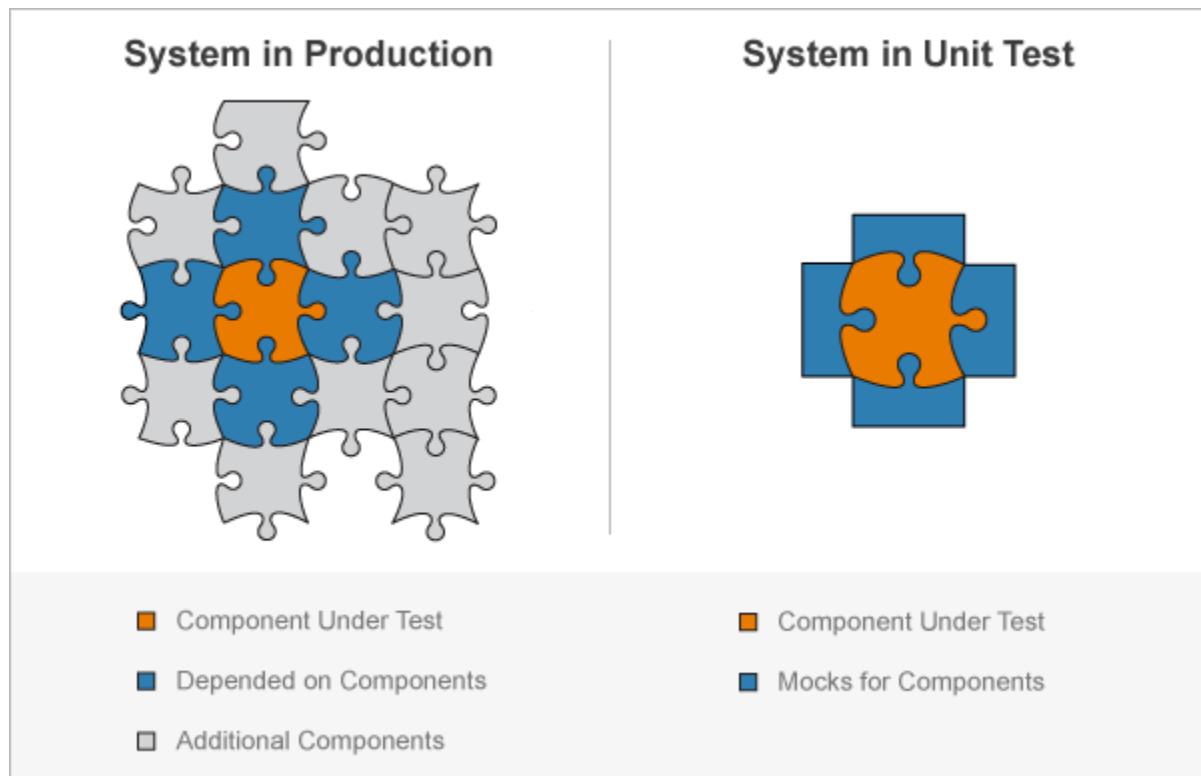
[keepMeasuring](#) | [runperf](#)

Related Examples

- “Overview of Performance Testing Framework” on page 34-164
- “Test Performance Using Classes” on page 34-172

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 *stubsing*). 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 component under test interacts with the mock object, for example, by calling a mock object method or accessing a mock object property. The mock object carries out predefined actions in response to these interactions.

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 "FOO", it returns the price yesterday as \$123 and anything before yesterday is \$234. Therefore, according to the `trader` function, the broker always buys stock "FOO". 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(...
    "FOO",yesterday),123);
testCase.assignOutputsWhen(dataServiceBehavior.lookupPrice(...
    "FOO",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("FOO",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 34-188.

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,"FOO",100)
trader(stubDataService,spyBroker,"FOO",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 FOO stock.

```
import matlab.mock.constraints.WasCalled;
testCase.verifyCalled(brokerBehavior.buy("FOO",100))
```

Verification passed.

Verify that FOO 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))
```

Verification passed.

Verify that the `buy` method was not called requesting 100 shares of the BAR stock.

```
testCase.verifyNotCalled(brokerBehavior.buy("BAR",100))
```

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	Duration
'TraderTest/buysStockWhenDrops'	true	false	false	0.24223
'TraderTest/doesNotBuyStockWhenIncreases'	true	false	false	0.073614

See Also

Specify Mock Object Behavior

In this section...

- “Define Mock Method Behavior” on page 34-188
- “Define Mock Property Behavior” on page 34-189
- “Define Repeating and Subsequent Behavior” on page 34-190
- “Summary of Behaviors” on page 34-192

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 34-181.

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 (line 368)
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 (line 346)
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 (line 346)
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 =
"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 (line 368)
The following property set was specified to throw an exception:
<Mock>.Name = "John"

```

```

mock.Name = "Penny"

```

```

Error using matlab.mock.internal.MockContext/createMockObject/mockPropertySetCallback (line 368)
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
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 34-181

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 34-181.

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
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	

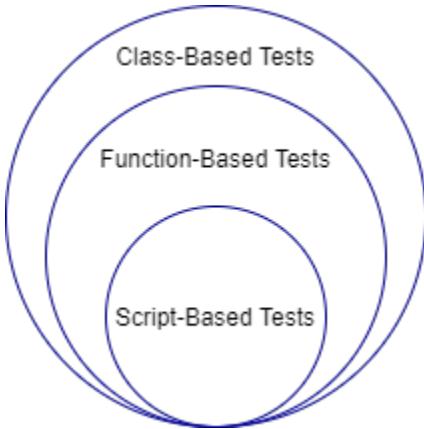
Type of Qualification	TestCase Method	matlab.mock.constraints Class	
		Use matlab.unittest.TestCase Method	With matlab.mock.constraints Class
Property was set a certain number of times	Not applicable	verifyThat, assumeThat, assertThat, or fatalAssertThat	WasSet
Property was set to a certain value	Not 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

Ways to Write Unit Tests

To guide software development and monitor for regressions in code functionality, you can write unit tests for your programs. The MATLAB unit testing framework supports three test authoring schemes:

- **Script-based unit tests:** Write each unit test as a separate section of a test script file. You can perform basic qualifications, access the diagnostics that the framework records on test results, refine the test suite by selecting the tests you want to run, and customize the test run by creating and configuring a `TestRunner` object.
- **Function-based unit tests:** Write each unit test as a local function within a test function file. Function-based tests subscribe to the xUnit testing philosophy. In addition to supporting the functionality provided by script-based tests, function-based tests give you access to a rich set of test authoring features. For example, you can use advanced qualification features, including constraints, tolerances, and test diagnostics.
- **Class-based unit tests:** Write each unit test as a `Test` method within a class definition file. In addition to supporting the functionality provided by script-based and function-based tests, class-based tests provide you with several advanced test authoring features and give you access to the full framework functionality. For example, you can use shared test fixtures, parameterize tests, and reuse test content.



Script-Based Unit Tests

With script-based tests, you can:

- Define variables to share among tests or preconditions necessary for tests.
- Perform basic qualifications using the `assert` function. For example, you can use `assert(isequal(actVal,expVal))` to assert that actual and expected values are equal. (Advanced qualification features are supported only for function-based and class-based tests.)
- Access test diagnostics recorded by the framework. For more information, see “Programmatically Access Test Diagnostics” on page 34-94. (Advanced diagnostic actions are supported only for function-based and class-based tests.)

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` elements. If you create an explicit test suite (using the `testsuite` function or a method of the `matlab.unittest.TestSuite` class), there are additional features available in script-based testing. With an explicit test suite, you can:

- Refine your suite, for example, using the classes in the `matlab.unittest.selectors` package. (Several of the selectors are applicable only for class-based tests.)
- Create a `TestRunner` object and customize it to run your tests. You can add the plugin classes in the `matlab.unittest.plugins` package to the test runner.

For more information about script-based tests, see “Write Script-Based Unit Tests” on page 34-6 and “Extend Script-Based Tests” on page 34-14.

Function-Based Unit Tests

Function-based tests support the functionality provided by script-based tests. In addition, with function-based tests, you can:

- Set up the pretest state of the system and return it to the original state after running the test. You can perform these tasks once per test file or once per unit test. For more information, see “Write Test Using Setup and Teardown Functions” on page 34-27.
- Use the fixture classes in the `matlab.unittest.fixtures` package (with the `applyFixture` method) to handle the setup and teardown of frequently used testing actions.
- Record diagnostic information at a certain verbosity level by using the `log` method.
- Use the full library of qualifications in the `matlab.unittest.qualifications` package. To determine which qualification to use, see “Table of Verifications, Assertions, and Other Qualifications” on page 34-45.
- Use advanced qualification features, including constraints, actual value proxies, tolerances, and test diagnostics. You can use the classes in the `matlab.unittest.constraints` and `matlab.unittest.diagnostics` packages in your qualifications.

For more information about function-based tests, see “Write Function-Based Unit Tests” on page 34-20 and “Extend Function-Based Tests” on page 34-32.

Class-Based Unit Tests

Class-based tests support the functionality provided by script-based and function-based tests. In addition, with class-based tests, you can:

- Use setup and teardown method blocks to implicitly set up the pretest environment state and return it to the original state after running the tests. For more information, see “Write Setup and Teardown Code Using Classes” on page 34-42.
- Share fixtures among classes. For more information, see “Write Tests Using Shared Fixtures” on page 34-51.
- Group tests into categories and then run the tests with specified tags. For more information, see “Tag Unit Tests” on page 34-47.
- Write parameterized tests to combine and execute tests on specified lists of parameters. For more information, see “Use Parameters in Class-Based Tests” on page 34-61.
- Use subclassing and inheritance to share and reuse test content. For example, you can reuse the parameters and methods defined in a test class by deriving subclasses. For more information, see “Hierarchies of Classes — Concepts”.

For more information about class-based tests, see “Author Class-Based Unit Tests in MATLAB” on page 34-36.

Extend Unit Testing Framework

The unit testing framework provides test tool authors the ability to extend test writing through custom constraints, diagnostics, fixtures, and plugins. For example, you can create a custom plugin and use it to extend the test runner when you run your script-based, function-based, or class-based unit tests. For more information, see “Extend Unit Testing Framework”.

See Also

More About

- “Write Script-Based Unit Tests” on page 34-6
- “Write Function-Based Unit Tests” on page 34-20
- “Author Class-Based Unit Tests in MATLAB” on page 34-36

Compile MATLAB Unit Tests

When you write tests using the MATLAB unit testing framework, you can create a standalone application (requires MATLAB Compiler) to run your tests on target machines that do not have MATLAB installed:

- To compile your MATLAB code, run the `compiler.build.standaloneApplication` or `mcc` command, or use the **Application Compiler** app.
- To run standalone applications, install the MATLAB Runtime. (If you use the **Application Compiler** app, you can decide whether to include the MATLAB Runtime installer in the generated application.) For more information, see “Install and Configure MATLAB Runtime” (MATLAB Compiler).

MATLAB Compiler supports only class-based unit tests. (You cannot compile script-based or function-based unit tests.) In addition, MATLAB Compiler currently does not support tests authored using the performance testing framework.

Run Tests with Standalone Applications

This example shows how to create a standalone application from your unit tests and run the generated application from a terminal window (on a Microsoft Windows platform).

In a file named `TestRand.m` in your current folder, create a parameterized test class that tests the MATLAB random number generator (see “`TestRand` Class Definition Summary” on page 34-203).

In your current folder, create the `runMyTests` function. This function creates a test suite from the `TestRand` class, runs the tests, and displays the test results.

```
function runMyTests
suite = matlab.unittest.TestSuite.fromClass(?TestRand);
runner = matlab.unittest.TestRunner.withNoPlugins;
results = runner.run(suite);
disp(results)
end
```

Compile the `runMyTests` function into a standalone application by running the `mcc` command in the Command Window. MATLAB Compiler generates an application in your current folder.

```
mcc -m runMyTests
```

Open a terminal window, navigate to the folder into which you packaged your standalone application, and run the application.

```
C:\work>runMyTests
1x1200 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details
```

```
Totals:
```

```
1200 Passed, 0 Failed, 0 Incomplete.
3.11 seconds testing time.
```

For more information on how to create and run standalone applications, see “Create Standalone Application from MATLAB” (MATLAB Compiler).

Run Tests in Parallel with Standalone Applications

Starting in R2020b, you can create standalone applications that support running tests in parallel (requires Parallel Computing Toolbox). This feature requires you to use the directive `%#function parallel.Pool` in the file that triggers the test run. The `%#function` pragma informs MATLAB Compiler that a `parallel.Pool` object must be included in the compilation to provide access to the parallel pool.

For example, consider the tests in the file `TestRand.m`. You can create a standalone application to run these tests in parallel by compiling this function.

```
function runMyTestsInParallel
%#function parallel.Pool
results = runtests('TestRand.m','UseParallel',true);
disp(results)
end
```

Compile the function into a standalone application using the `mcc` command. To instruct MATLAB Compiler to include the test file in the application, specify the file name using the `-a` option.

```
mcc -m runMyTestsInParallel -a TestRand.m
```

TestRand Class Definition Summary

This code provides the complete contents of the `TestRand` class.

```
classdef TestRand < matlab.unittest.TestCase
    properties (TestParameter)
        dim1 = createDimensionSizes;
        dim2 = createDimensionSizes;
        dim3 = createDimensionSizes;
        type = {'single','double'};
    end

    methods (Test)
        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
        function testClass(testCase,dim1,dim2,type)
            testCase.verifyClass(rand(dim1,dim2,type),type)
        end
    end
end

function sizes = createDimensionSizes
% Create logarithmically spaced sizes up to 100
```

```
sizes = num2cell(round(logspace(0,2,10)));
end
```

See Also

[compiler.build.standaloneApplication](#) | [mcc](#) | [run \(TestRunner\)](#) | [run \(TestSuite\)](#) | [runInParallel](#) | [runtests](#)

More About

- “Ways to Write Unit Tests” on page 34-199
- “Create Standalone Application from MATLAB” (MATLAB Compiler)

Develop and Integrate Software with Continuous Integration

Continuous integration (CI) is the practice of integrating code changes into a shared repository on a frequent basis. It improves team throughput and software quality by automating and standardizing activities such as building code, testing, and packaging. For example, each time a developer pushes new committed changes to the remote repository, the continuous integration platform can automatically run a suite of tests to ensure that the changes do not cause any conflicts in the target branch of the remote repository.

The benefits of continuous integration include:

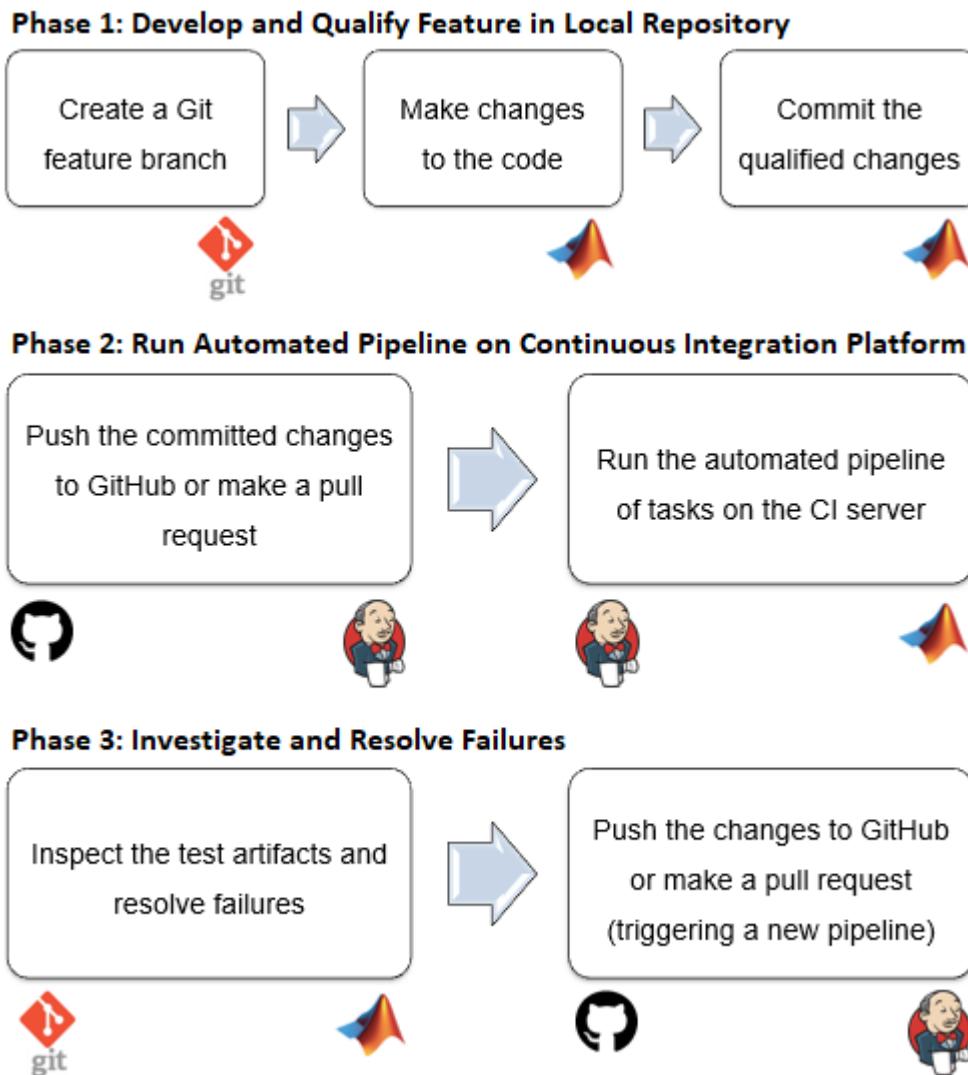
- Finding problems in software and fixing them soon after they are introduced.
- Adding more features while reducing the resources required for debugging code.
- Minimizing integration and deployment overheads by performing integration on a continuous basis.
- Clearly communicating the state of software and the changes that have been made to it.

Continuous Integration Workflow

A typical software development workflow using continuous integration involves several steps:

- 1 Cloning the remote repository and creating a feature branch
- 2 Editing files and committing the changes to the local repository
- 3 Pushing the committed changes to the remote repository (which triggers an automated pipeline of tasks such as compiling MEX files, packaging toolboxes, and testing on the CI platform)
- 4 Analyzing the reports generated by the CI platform and fixing the errors within the pipeline
- 5 Merging the remote feature branch into the main branch through a pull request (which triggers another automated pipeline of tasks on the CI platform)
- 6 Analyzing the reports generated by the CI platform and resolving the merge failures

This figure shows an example of the development cycle using the Jenkins™ CI server and open-source source code management tools such as Git and GitHub. For information on how to interface MATLAB with Jenkins, see Run MATLAB Tests on Jenkins Server.



Phase 1: Develop and Qualify Feature in Local Repository

Develop a feature and commit your changes to the local repository:

- 1 Clone the GitHub repository and create a new feature branch.
- 2 Make changes to the existing files or add new files as appropriate.
- 3 Run MATLAB and Simulink tests to qualify the changes and commit them to the local repository.

Phase 2: Run Automated Pipeline on Continuous Integration Platform

Run an automated pipeline of tasks (including testing) when you push your changes to the remote repository or when you make a pull request:

- 1 Trigger an automated pipeline of tasks on Jenkins by pushing the committed changes to GitHub or by making a pull request to merge the remote feature branch into the main branch.
- 2 Jenkins runs the automated pipeline, including MATLAB and Simulink tests, and generates artifacts as specified in the project configuration.

Phase 3: Investigate and Resolve Failures

If you do not succeed in pushing your changes or making a pull request, follow these steps:

- 1 Inspect the automated pipeline results and the generated test artifacts. Make appropriate changes to your code.
- 2 Trigger a new pipeline on Jenkins by pushing your changes to GitHub or by making a pull request.

Integration engineers can use Jenkins test artifacts to decide when to merge the feature branch into the main branch.

Continuous Integration with MathWorks Products

You can perform continuous integration with MATLAB on various continuous integration platforms. You can run and test your MATLAB code and Simulink models, generate artifacts, and publish your results to the platforms. For more information, see “Continuous Integration with MATLAB on CI Platforms” on page 34-213.

In addition to MATLAB, different toolboxes support continuous integration workflows. This table lists common continuous integration use cases for models and code.

Toolbox	Use Case	More Information
Simulink	<ul style="list-style-type: none"> • Build and test models and projects • Cache files for simulation and code generation 	“About Source Control with Projects” (Simulink) “Using a Project with Git” (Simulink) “Share Simulink Cache Files for Faster Simulation” (Simulink)
Simulink Test™	Run test files on CI platforms and collect CI-compatible coverage using Simulink Coverage™	“Continuous Integration” (Simulink Test)
Simulink Check™	Use Jenkins to detect metric threshold violations in a model	“Fix Metric Threshold Violations in a Continuous Integration Systems Workflow” (Simulink Check)
Simulink Requirements™	Summarize requirements verification results for tests run on CI platforms	“Include Results from External Sources in Verification Status” (Simulink Requirements)

Toolbox	Use Case	More Information
Polyspace® Bug Finder™ Server™, Polyspace Code Prover™ Server	<ul style="list-style-type: none"> Run a Polyspace analysis on C/C++ code as part of continuous integration, for instance with Jenkins Upload the analysis results (bugs, run-time errors, or coding standard violations) for review in the Polyspace Access web interface Send e-mail notifications with Polyspace Bug Finder or Polyspace Code Prover results 	<p>“Run Bug Finder Analysis on Server During Continuous Integration” (Polyspace Bug Finder Server)</p> <p>“Run Code Prover Analysis on Server During Continuous Integration” (Polyspace Code Prover Server)</p>

See Also

`matlab.unittest.plugins` Package

More About

- “Set Up Git Source Control” on page 33-23
- “Use Source Control with Projects” on page 32-45
- “Explore an Example Project” on page 32-63
- “Continuous Integration with MATLAB on CI Platforms” on page 34-213

External Websites

- MathWorks Blogs: Developer Zone - Continuous Integration

Generate Artifacts Using MATLAB Unit Test Plugins

The MATLAB® unit testing framework enables you to customize your test runner using the plugin classes in the `matlab.unittest.plugins` package. You can use some of these plugin classes to generate test reports and artifacts compatible with continuous integration (CI) platforms:

- `matlab.unittest.plugins.TestReportPlugin` creates a plugin that directs the test runner to produce a test result report. Using this plugin, you can produce readable and archivable test reports.
- `matlab.unittest.plugins.TAPPlugin` creates a plugin that produces a Test Anything Protocol (TAP) stream.
- `matlab.unittest.plugins.XMLPlugin` creates a plugin that produces JUnit-style XML output.
- `matlab.unittest.plugins.CodeCoveragePlugin` creates a plugin that produces a line coverage report for MATLAB source code.

You also can generate CI-compatible artifacts when you run Simulink® Test™ test cases. For more information, see “Output Results for Continuous Integration Systems” (Simulink Test).

Run Tests with Customized Test Runner

This example shows how to create a test suite and customize the test runner to report on test run progress and produce CI-compatible artifacts.

In a file in your current folder, create the function `quadraticSolver`, which returns the roots of quadratic polynomials.

```
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
```

To test `quadraticSolver`, create the test class `SolverTest` in your current folder.

```
classdef SolverTest < matlab.unittest.TestCase
methods(Test)
    function realSolution(testCase)
        actSolution = quadraticSolver(1,-3,2);
        expSolution = [2,1];
        testCase.verifyEqual(actSolution,expSolution)
    end
    function imaginarySolution(testCase)
        actSolution = quadraticSolver(1,2,10);
        expSolution = [-1+3i, -1-3i];
        testCase.verifyEqual(actSolution,expSolution)
    end
    function nonnumericInput(testCase)
```

```
    testCase.verifyError(@()quadraticSolver(1,'-3',2), ...
        'quadraticSolver:InputMustBeNumeric')
    end
end
end
```

At the command prompt, create a test suite from the `SolverTest` class.

```
suite = testsuite('SolverTest');
```

Create a `TestRunner` instance that produces output using the `matlab.unittest.TestRunner.withTextOutput` method. This method enables you to set the maximum verbosity level for logged diagnostics and the display level for test event details. In this example, the test runner displays test run progress at the `matlab.unittest.Verbose.Detailed` level (level 3).

```
import matlab.unittest.TestRunner
runner = TestRunner.withTextOutput('OutputDetail',3);
```

Create a `TestReportPlugin` instance that sends output to the file `testreport.pdf` and add the plugin to the test runner.

```
import matlab.unittest.plugins.TestReportPlugin
pdfFile = 'testreport.pdf';
p1 = TestReportPlugin.producingPDF(pdfFile);
runner.addPlugin(p1)
```

Create an `XMLPlugin` instance that writes JUnit-style XML output to the file `junittestresults.xml`. Then, add the plugin to the test runner.

```
import matlab.unittest.plugins.XMLPlugin
xmlFile = 'junittestresults.xml';
p2 = XMLPlugin.producingJUnitFormat(xmlFile);
runner.addPlugin(p2)
```

Create a plugin that outputs a Cobertura code coverage report for the source code in the file `quadraticSolver.m`. Instruct the plugin to write its output to the file `cobertura.xml` and add the plugin to the test runner.

```
import matlab.unittest.plugins.CodeCoveragePlugin
import matlab.unittest.plugins.codecoverage.CoberturaFormat
sourceCodeFile = 'quadraticSolver.m';
reportFile = 'cobertura.xml';
reportFormat = CoberturaFormat(reportFile);
p3 = CodeCoveragePlugin.forFile(sourceCodeFile, 'Producing', reportFormat);
runner.addPlugin(p3)
```

Run the tests.

```
results = runner.run(suite)

Running SolverTest
Setting up SolverTest
Done setting up SolverTest in 0 seconds
Running SolverTest/realSolution
Done SolverTest/realSolution in 0.89958 seconds
Running SolverTest/imaginarySolution
Done SolverTest/imaginarySolution in 0.021852 seconds
```

```

Running SolverTest/nonnumericInput
Done SolverTest/nonnumericInput in 0.40315 seconds
Tearing down SolverTest
Done tearing down SolverTest in 0 seconds
Done SolverTest in 1.3246 seconds

Generating test report. Please wait.
Preparing content for the test report.

Adding content to the test report.
Writing test report to file.
Test report has been saved to:
C:\TEMP\Bdoc21a_1606923_5032\ib8F3FCD\17\tpafa2a539\matlab-ex57066699\testreport.pdf

results =
1x3 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
  3 Passed, 0 Failed, 0 Incomplete.
  1.3246 seconds testing time.

```

List the files in your current folder. The three specified artifacts are stored in your current folder.

```

dir

.

.

GenerateArtifactsUsingMATLABUnitTestPluginsExample mlx
SolverTest.m
cobertura.xml
junittestresults.xml
quadraticSolver.m
testreport.pdf

```

You can process the generated artifacts on CI platforms. You also can view the contents of the generated artifacts. For example, open the PDF test report.

```
open('testreport.pdf')
```

See Also

[matlab.unittest.TestRunner](#) | [matlab.unittest.plugins Package](#) |
[matlab.unittest.plugins.CodeCoveragePlugin](#) | [matlab.unittest.plugins.TAPPlugin](#) |
[matlab.unittest.plugins.TestReportPlugin](#) | [matlab.unittest.plugins.XMLPlugin](#)

More About

- “Develop and Integrate Software with Continuous Integration” on page 34-205

- “Continuous Integration with MATLAB on CI Platforms” on page 34-213
- “Output Results for Continuous Integration Systems” (Simulink Test)

Continuous Integration with MATLAB on CI Platforms

You can use different continuous integration (CI) platforms to run MATLAB code and Simulink models as part of your automated pipeline of tasks. In addition, the MATLAB unit testing framework enables you to create a test suite and a test runner, and customize your test runner for continuous integration workflows with the plugin classes in the `matlab.unittest.plugins` package.

To facilitate running and testing software with continuous integration, MATLAB seamlessly integrates with several CI platforms, such as Azure® DevOps, CircleCI®, and Jenkins. You can use these platforms to:

- Run MATLAB scripts, functions, and statements in your pipeline.
- Run MATLAB and Simulink tests and generate artifacts, such as JUnit test results and Cobertura code coverage reports.

Depending on the CI platform, you might:

- Configure your pipeline using a script or user interface.
- Set up the platform to run MATLAB on premises or in the cloud.

Azure DevOps

To perform continuous integration with MATLAB on Azure DevOps, install an extension to your Azure DevOps organization. To run MATLAB in your pipeline, use the extension to author your pipeline YAML in a file named `azure-pipelines.yml` in the root of your repository. You can run your pipeline using a Linux agent in the cloud or a self-hosted agent. For more information, see the extension on Visual Studio Marketplace.

CircleCI

To perform continuous Integration with MATLAB on CircleCI, opt-in to using third-party orbs in your organization security settings. To run MATLAB in your pipeline, import the appropriate orb to author your pipeline YAML in a file named `.circleci/config.yml` in the root of your repository. You can run your pipeline using a Linux machine executor in the cloud. For more information, see the orb on CircleCI Orb Registry.

GitHub Actions

To perform continuous integration with MATLAB on GitHub Actions, make sure GitHub Actions is enabled for your repository. To run MATLAB in your workflow, use the appropriate actions when you define your workflow in the `.github/workflows` directory of your repository. You can run your workflow using a Linux runner in the cloud or a self-hosted runner. For more information, see Use MATLAB with GitHub Actions.

Jenkins

To perform continuous integration with MATLAB on Jenkins, install a plugin on your Jenkins agent. Then, you can use an interface to run MATLAB in freestyle and multi-configuration (matrix) projects. You also can configure your pipeline as code checked into source control. For more information, see the plugin on Jenkins Plugins Index.

Travis CI

To perform continuous integration with MATLAB on Travis CI, specify the MATLAB language when you author your pipeline YAML in a file named `.travis.yml` in the root of your repository. You can run your pipeline using a Linux agent in the cloud. For more information, see the language in Travis CI Documentation.

Other Platforms

To perform continuous integration with MATLAB on other CI platforms, use the `matlab` command with the `-batch` option in your pipeline. You can use `matlab -batch` to run MATLAB scripts, functions, and statements noninteractively. For example, `matlab -batch "myscript"` starts MATLAB noninteractively and runs the commands in a file named `myscript.m`. MATLAB terminates automatically with exit code 0 if the specified script, function, or statement executes successfully without error. Otherwise, MATLAB terminates with a nonzero exit code.

See Also

`matlab.unittest.plugins` Package

More About

- “Develop and Integrate Software with Continuous Integration” on page 34-205
- “Generate Artifacts Using MATLAB Unit Test Plugins” on page 34-209

System object Usage and Authoring

- “What Are System Objects?” on page 35-2
- “System Objects vs MATLAB Functions” on page 35-5
- “System Design in MATLAB Using System Objects” on page 35-7
- “Define Basic System Objects” on page 35-11
- “Change the Number of Inputs” on page 35-13
- “Validate Property and Input Values” on page 35-16
- “Initialize Properties and Setup One-Time Calculations” on page 35-18
- “Set Property Values at Construction Time” on page 35-20
- “Reset Algorithm and Release Resources” on page 35-22
- “Define Property Attributes” on page 35-24
- “Hide Inactive Properties” on page 35-26
- “Limit Property Values to Finite List” on page 35-28
- “Process Tuned Properties” on page 35-32
- “Define Composite System Objects” on page 35-34
- “Define Finite Source Objects” on page 35-36
- “Save and Load System Object” on page 35-38
- “Define System Object Information” on page 35-41
- “Handle Input Specification Changes” on page 35-43
- “Summary of Call Sequence” on page 35-45
- “Detailed Call Sequence” on page 35-48
- “Tips for Defining System Objects” on page 35-50
- “Insert System Object Code Using MATLAB Editor” on page 35-53
- “Analyze System Object Code” on page 35-58
- “Use Global Variables in System Objects” on page 35-60
- “Create Moving Average System object” on page 35-64
- “Create New System Objects for File Input and Output” on page 35-69
- “Create Composite System object” on page 35-75

What Are System Objects?

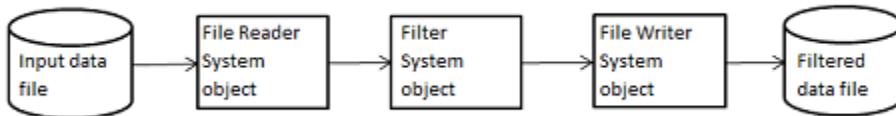
In this section...

["Running a System Object" on page 35-3](#)

["System Object Functions" on page 35-3](#)

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, `asyobj()`.

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 <code>step</code>	<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.
<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> .

See Also

`matlab.System`

Related Examples

- “System Objects vs MATLAB Functions” on page 35-5
- “System Design in MATLAB Using System Objects” on page 35-7
- “System Design in Simulink Using System Objects” (Simulink)

System Objects vs MATLAB Functions

In this section...

- “System Objects vs. MATLAB Functions” on page 35-5
- “Process Audio Data Using Only MATLAB Functions Code” on page 35-5
- “Process Audio Data Using System Objects” on page 35-6

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.

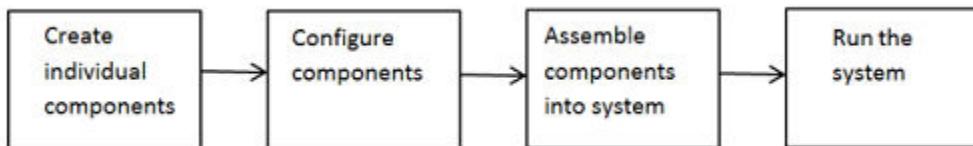
System Design in MATLAB Using System Objects

In this section...

- “System Design and Simulation in MATLAB” on page 35-7
- “Create Individual Components” on page 35-7
- “Configure Components” on page 35-8
- “Create and Configure Components at the Same Time” on page 35-8
- “Assemble Components Into System” on page 35-9
- “Run Your System” on page 35-9
- “Reconfiguring Objects” on page 35-10

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 35-7 — Create the System objects to use in your system. “Create Individual Components” on page 35-7. 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 35-8 — 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 35-8.
- 3 “Assemble Components Into System” on page 35-9 — 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 35-9.
- 4 “Run Your System” on page 35-9 — Run your program. You can change tunable properties while your system is running. See “Run Your System” on page 35-9 and “Reconfiguring Objects” on page 35-10.

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 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 35-10 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 low pass 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 low pass 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 nargin on any System object. For information on all available System object functions, see “System Object Functions” on page 35-3.

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 35-13.

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 35-13
- “System Design and Simulation in MATLAB” on page 35-7

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
```

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 `getNumInputsImpl` method 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
```

```
    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 35-16
- “Define Basic System Objects” on page 35-11
- “Using ~ as an Input Argument in Method Definitions” on page 35-50

Validate Property and Input Values

This example shows how to verify that the inputs and property values given to your System object are valid.

Validate a Single Property

To validate a property value, independent of other properties, use MATLAB class property validation. This example shows how to specify a logical property, a positive integer property, and a string property that must be one of three values.

```
properties
    UseIncrement (1,1) logical = false
    WrapValue (1,1) {mustBePositive, mustBeInteger} = 1
    Color (1,1) string {mustBeMember(Color, ["red","green","blue"])} = "red"
end
```

Validate Interdependent Properties

To validate the values of two or more interdependent properties, use the `validatePropertiesImpl`. This example shows how to write `validatePropertiesImpl` to verify that a logical property (`UseIncrement`) is `true` and the value of `WrapValue` is larger than `Increment`.

```
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
```

Validate Inputs

To validate input values, use the `validateInputsImpl` method. 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
```

Complete Class Example

This example is a complete System object that shows examples of each type of validation syntax.

```
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
```

```

UseIncrement (1,1) logical = false
WrapValue (1,1) {mustBePositive, mustBeInteger} = 10
Increment (1,1) {mustBePositive, mustBeInteger} = 1
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

```

See Also

[validateInputsImpl](#) | [validatePropertiesImpl](#)

Related Examples

- “Define Basic System Objects” on page 35-11
- “Change Input Complexity, Dimensions, or Data Type” on page 35-10
- “Summary of Call Sequence” on page 35-45
- “Property Set Methods”
- “Using ~ as an Input Argument in Method Definitions” on page 35-50

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
```

See Also

[matlab.System Constructor](#) | [releaseImpl](#) | [setupImpl](#) | [stepImpl](#)

Related Examples

- “Release System Object Resources” on page 35-22
- “Define Property Attributes” on page 35-24
- “Summary of Call Sequence” on page 35-45

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

    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

        % This System object writes the input to the file.
    end
```

```
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 35-24
- “Release System Object Resources” on page 35-22

Reset Algorithm and Release Resources

In this section...

- ["Reset Algorithm State" on page 35-22](#)
- ["Release System Object Resources" on page 35-22](#)

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 `release` is called on a System object, the internal `releaseImpl` method is called if `step` or `setup` was previously called (see “Summary of Call Sequence” on page 35-45). 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 35-29.

See Also

`releaseImpl` | `resetImpl`

More About

- “Summary of Call Sequence” on page 35-45
- “Initialize Properties and Setup One-Time Calculations” on page 35-18

Define 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 and property validation, System objects can use `Nontunable` or `DiscreteState`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

By default all properties are *tunable*, meaning the value of the property can change at any time.

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.

When you use the System object, you can only change nontunable properties before calling the object or after calling the `release` function. For example, you define the `InitialValue` property as nontunable and set its value to 0.

```
properties (Nontunable)
    InitialValue = 0;
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 35-45)

For example, you define the `Count` property as a discrete state:

```
properties (DiscreteState)
    Count;
end
```

Example Class with Various Property Attributes

This example shows two nontunable properties, a discrete state property, and also MATLAB class property validation to set 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
    % The maximum value of the counter, must be a positive integer scalar
    maxValue (1, 1) {mustBePositive, mustBeInteger} = 3
end

properties
    % Whether to increment the counter, must be a logical scalar
    increment (1, 1) logical = 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

```

See Also

More About

- “Class Attributes”
- “Property Attributes”
- “What You Cannot Change While Your System Is Running” on page 35-9
- “Summary of Call Sequence” on page 35-45

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 of a dependent 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

% 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
```

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 can either use enumerations or property validation.

For a System object that is used in a MATLAB System block in Simulink you can use enumerations or property validation. If you use enumerations, 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).

Property Validation with `mustBeMember`

To limit property values with property validation, you use the `mustBeMember` validation function.

This example defines a `Style` property that can have the values `solid`, `dash`, or `dot`. The default value is `solid` and the `(1,1)` defines the property as a scalar.

```
properties
    Style (1,1) string {mustBeMember(Style, ["solid", "dash", "dot"])} = "solid";
end
```

To support case-insensitive match, use `matlab.system.mustBeMember` instead.

```
properties
    Style (1,:) char {matlab.system.mustBeMember(Style, ["solid", "dash", "dot"])} = "solid";
end
```

Enumeration Property

To use enumerated data in a System object, you refer to the enumerations as properties in your System object class definition and define your enumerated class in a separate class definition file.

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.

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 on page 1-4.

In the System object, the `Color` property is defined as a `ColorValues` object with `blue` as the default. The `(1,1)` defines the `Color` property as a scalar:

```

properties
    Color (1, 1) ColorValues = ColorValues.blue
end

```

Create a Whiteboard System object

This example shows the class definition of a Whiteboard System object™, two types of finite list properties, and how to use the object. Each time you run the whiteboard object, it draws a line on a whiteboard.

Definition of the Whiteboard System object

```

type Whiteboard.m
classdef Whiteboard < matlab.System
    % Whiteboard Draw lines on a figure window
    %

properties(Nontunable)
    Color (1, 1) ColorValues = ColorValues.blue
    Style (1,1) string {mustBeMember(Style, ["solid","dash","dot"])} = "solid";
end

methods (Access = protected)
    function stepImpl(obj)
        h = Whiteboard.getWhiteboard();
        switch obj.Style
            case "solid"
                linestyle = "-";
            case "dash"
                linestyle = "--";
            case "dot"
                linestyle = ":";
        end
        plot(h, randn([2,1]), randn([2,1]), ...
            "Color",string(obj.Color), "LineStyle",linestyle);
    end

    function releaseImpl(~)
        cla(Whiteboard.getWhiteboard());
        hold on
    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

```

Construct the System object.

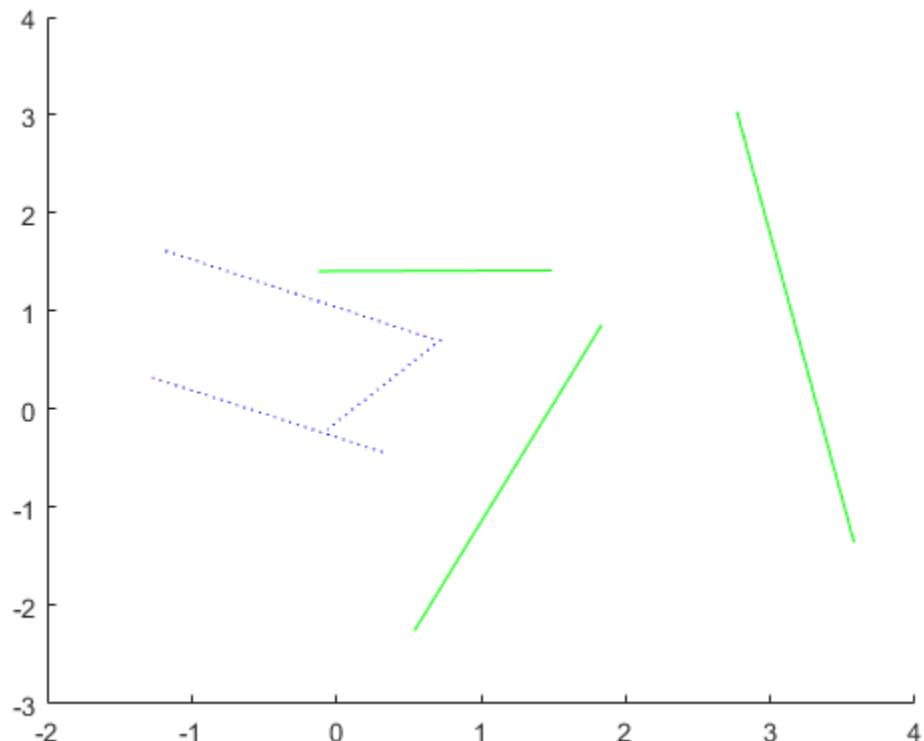
```
greenInk = Whiteboard;
blueInk = Whiteboard;
```

Change the color and set the blue line style.

```
greenInk.Color = "green";
blueInk.Color = "blue";
blueInk.Style = "dot";
```

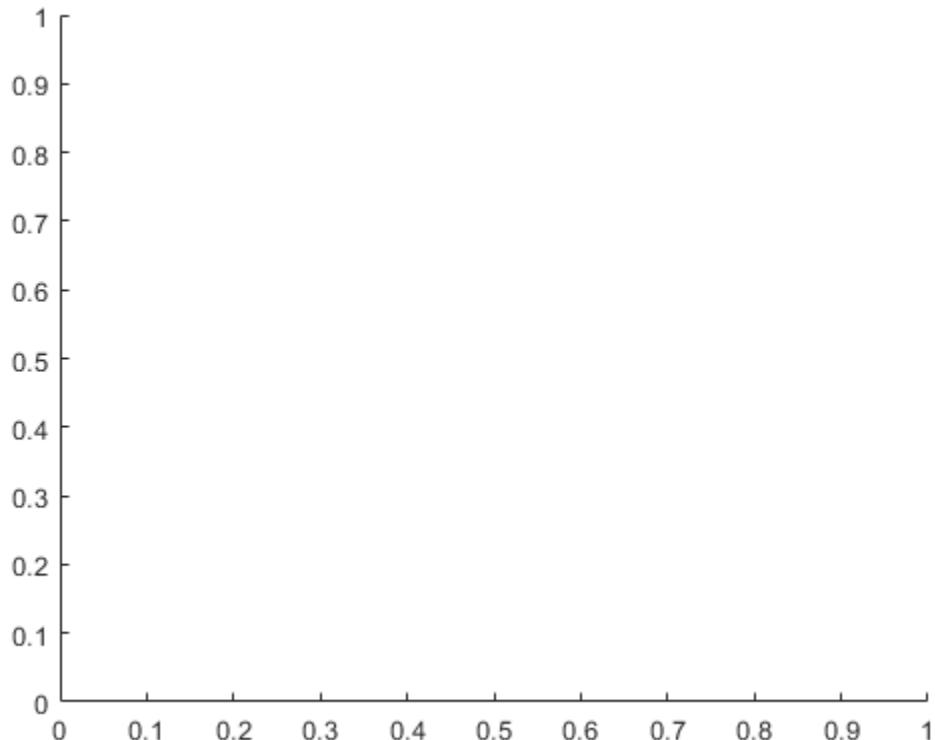
Draw a few lines.

```
for i=1:3
    greenInk();
    blueInk();
end
```



Clear the whiteboard

```
release(greenInk);
release(blueInk);
```



See Also

Related Examples

- “Validate Property Values”
- “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
```

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);
            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 35-36](#)

["Complete Class Definition File with Finite Source" on page 35-36](#)

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 35-50

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

        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

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

```

```
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

    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

        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
```

```
    else
        s = struct('Count',obj.Count);
    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)
    isComplex (1,1) logical = 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 Counter System 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

    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 35-9

Summary of Call Sequence

In this section...

- “Setup Call Sequence” on page 35-45
- “Running the Object or Step Call Sequence” on page 35-45
- “Reset Method Call Sequence” on page 35-46
- “Release Method Call Sequence” on page 35-47

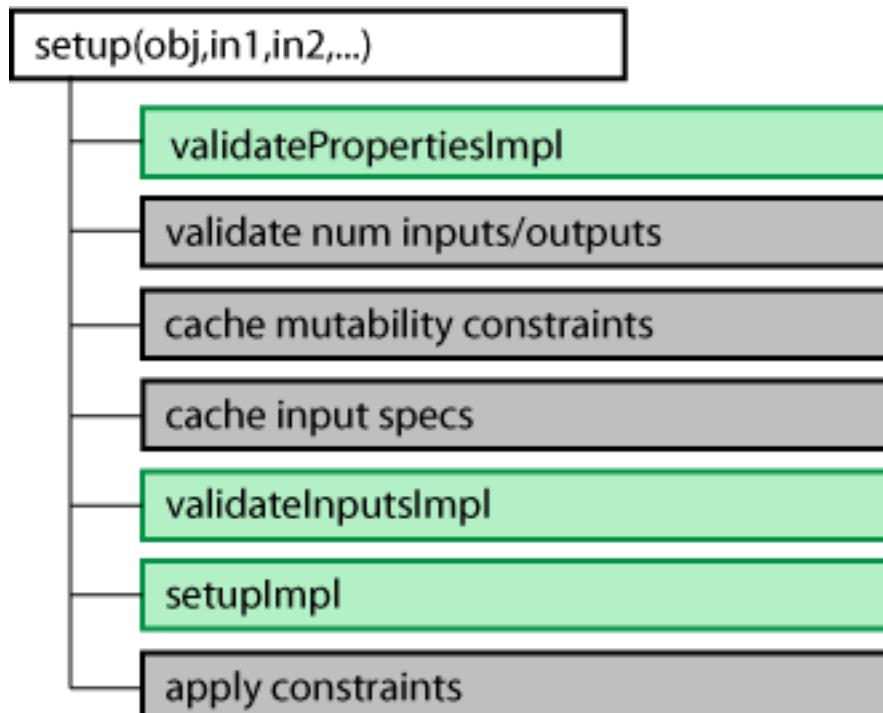
The diagrams show an abstract view of which actions are performed when you call a System object. The background color of each action indicates the type of call.

- 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 35-48.

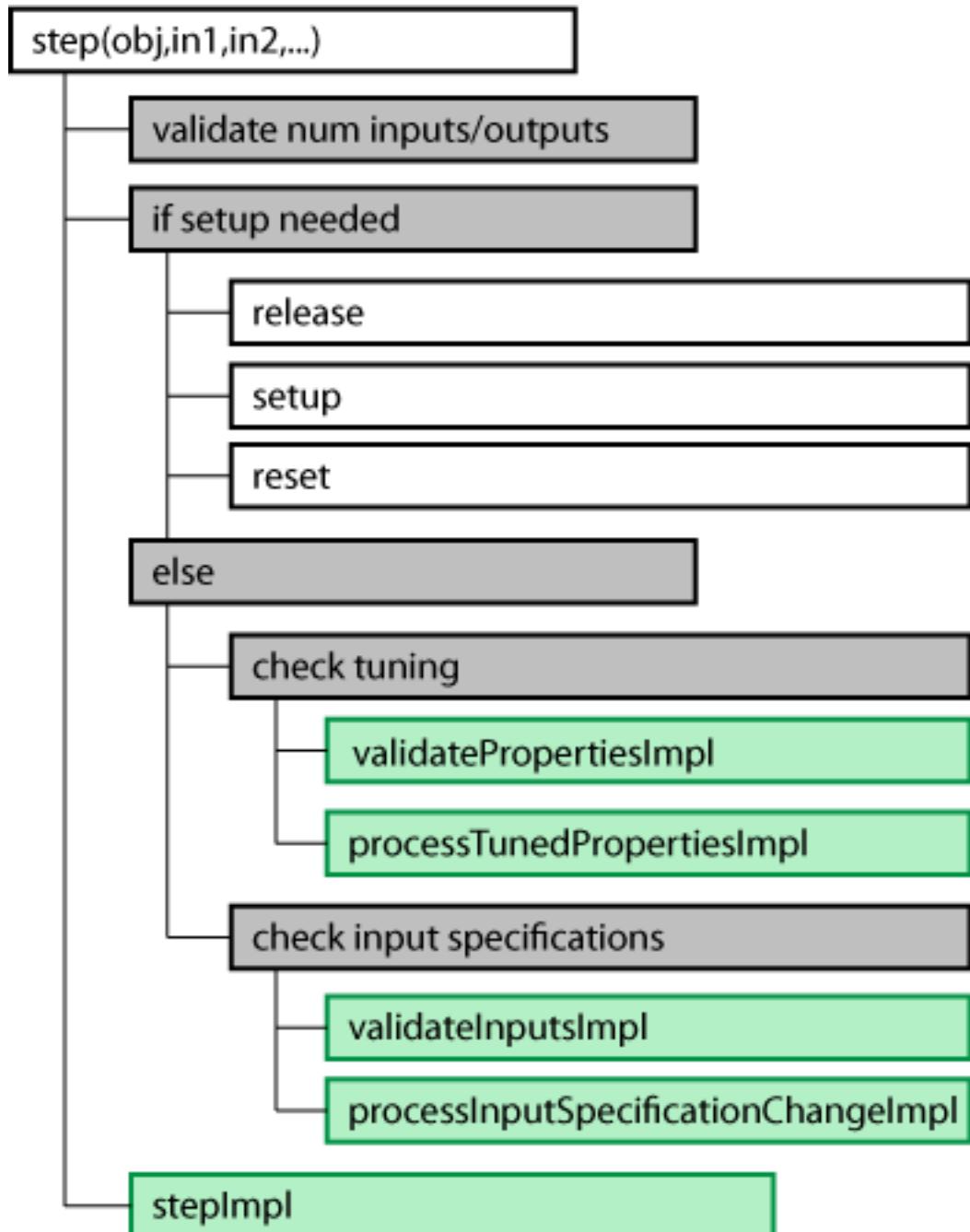
Setup Call Sequence

This hierarchy shows the actions performed when you call the `setup` function.



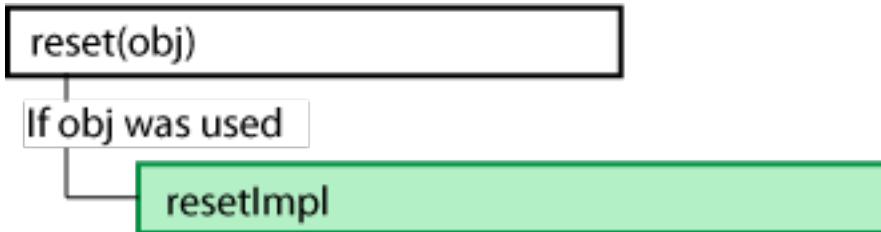
Running the Object or Step Call Sequence

This hierarchy shows the actions performed when you call the `step` function.



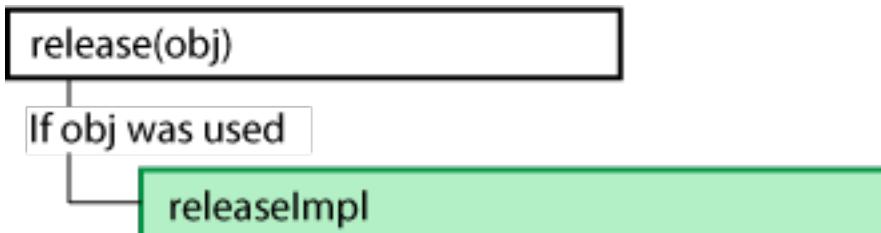
Reset Method Call Sequence

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 35-22
- “Reset Algorithm State” on page 35-22
- “Set Property Values at Construction Time” on page 35-20
- “Define Basic System Objects” on page 35-11

Detailed Call Sequence

In this section...

["setup Call Sequence" on page 35-48](#)

["Running the Object or step Call Sequence" on page 35-48](#)

["reset Call Sequence" on page 35-49](#)

["release Call Sequence" on page 35-49](#)

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 35-45](#).

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 35-49](#)
- 2 `validatePropertiesImpl`
- 3 `isDiscreteStateSpecificationMutableImpl`
- 4 `isInputDataTypeMutableImpl`
- 5 `isInputComplexityMutableImpl`
- 6 `isInputSizeMutableImpl`
- 7 `isTunablePropertyDataTypeMutableImpl`
- 8 `validateInputsImpl`
- 9 If the System object uses nondirect feedthrough methods, 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 35-47](#)
- b `setup` on [page 35-48](#)
- c `reset` on [page 35-49](#)

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 35-22
- “Reset Algorithm State” on page 35-22
- “Set Property Values at Construction Time” on page 35-20
- “Define Basic System Objects” on page 35-11

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 `processInputSpecificationChangeImpl`. 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 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 35-53](#)

["Create a Temperature Enumeration" on page 35-55](#)

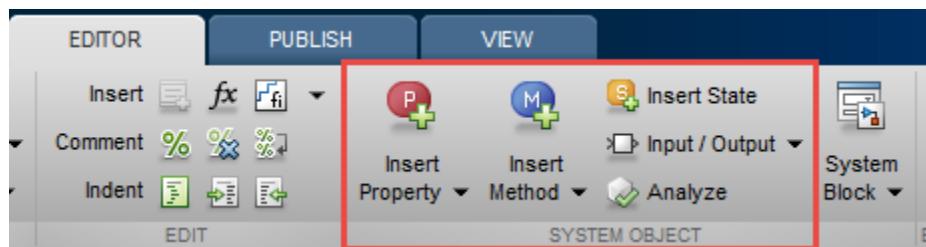
["Create Custom Property for Freezing Point" on page 35-56](#)

["Add Method to Validate Inputs" on page 35-57](#)

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.

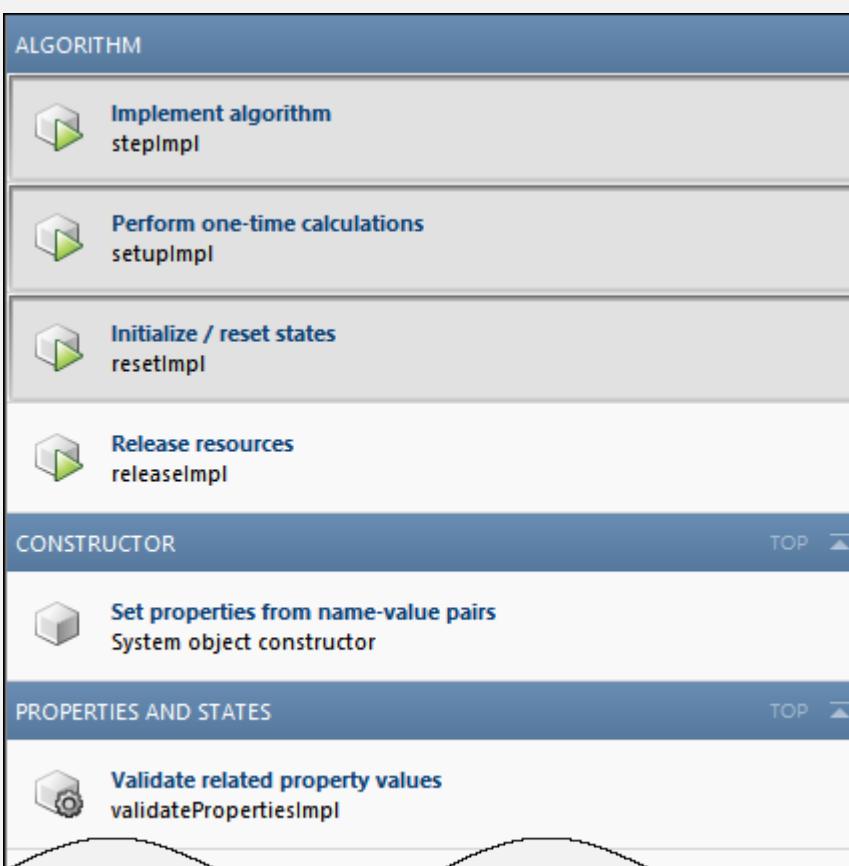


To add predefined code to your System object, select the code from the appropriate menu. For example, when you click **Insert Property > Numeric**, the MATLAB Editor adds the following code:

```
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

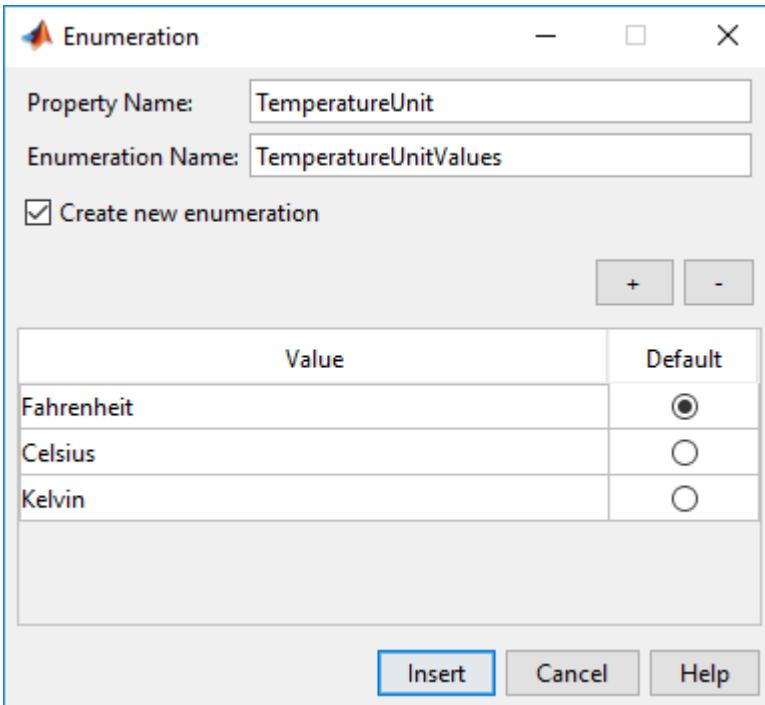
Properties	Properties of the System object: Numeric, Logical, Enumeration, Positive Integer, Tunable Numeric, Private, Protected, and Custom. When you select Enumeration or Custom Properties, a separate dialog box opens to guide you in creating these properties.
Methods	<p>Methods commonly used in System 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>The screenshot shows the MATLAB Editor's Insert Method menu. The 'Release resources' option is highlighted with a blue background, indicating it is present in the current code. Other options visible include 'Implement algorithm', 'Perform one-time calculations', 'Initialize / reset states', and 'Set properties from name-value pairs'.</p>
States	Properties containing the <code>DiscreteState</code> 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><code>function y = stepImpl(obj,u,u2) % Implement algorithm. Calculate y as a function of % input u and discrete states. y = u; end</code></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** check box.
- 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
```

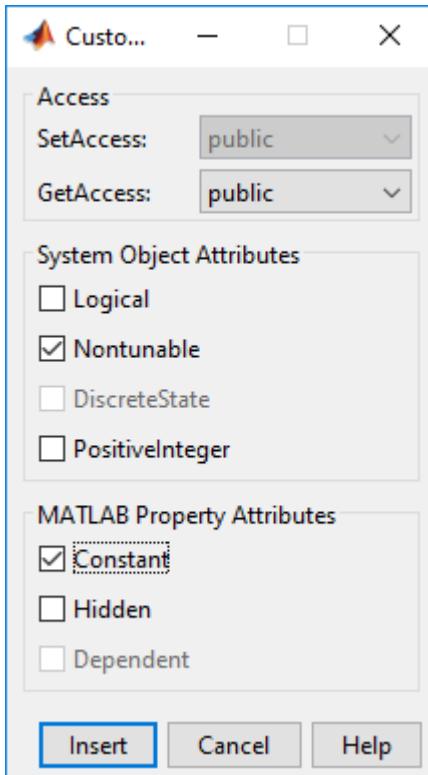
In the System object class definition, the following code was added:

```
properties(Nontunable)
    TemperatureUnit (1, 1) TemperatureUnitValues = TemperatureUnitValues.Fahrenheit
end
```

For more information on enumerations, see “Limit Property Values to Finite List” on page 35-28.

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 35-58

Analyze System Object Code

In this section...

"View and Navigate System object Code" on page 35-58

"Example: Go to StepImpl Method Using Analyzer" on page 35-58

View and Navigate System object Code

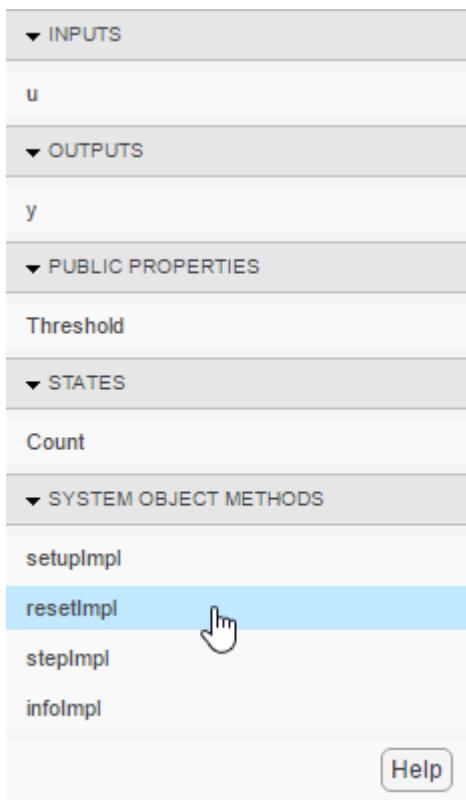
View and navigate System object code using the Analyzer.

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 35-53

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 20-12).

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.

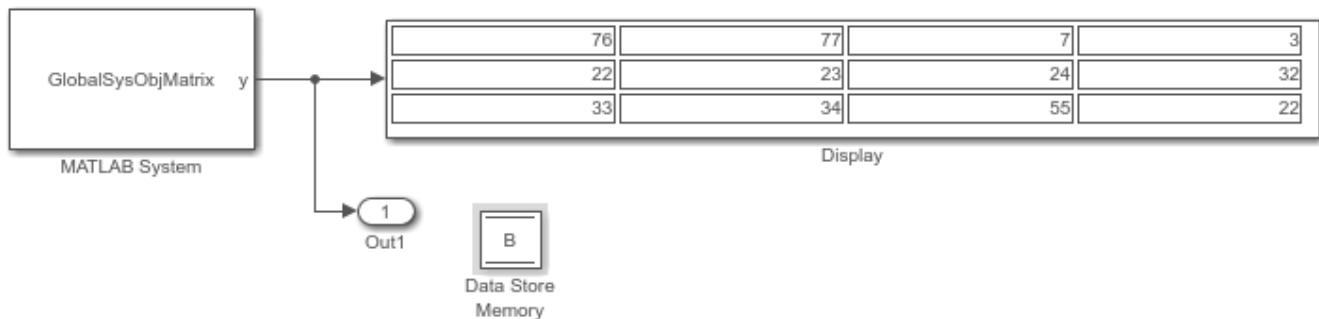
You set up and use global variables for the MATLAB System block in the same way as you do for the MATLAB Function block (see “Data Stores” (Simulink) and “Share Data Globally” (Simulink)). Like the MATLAB Function block, you must also use variable name matching with a Data Store Memory block to use global 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.

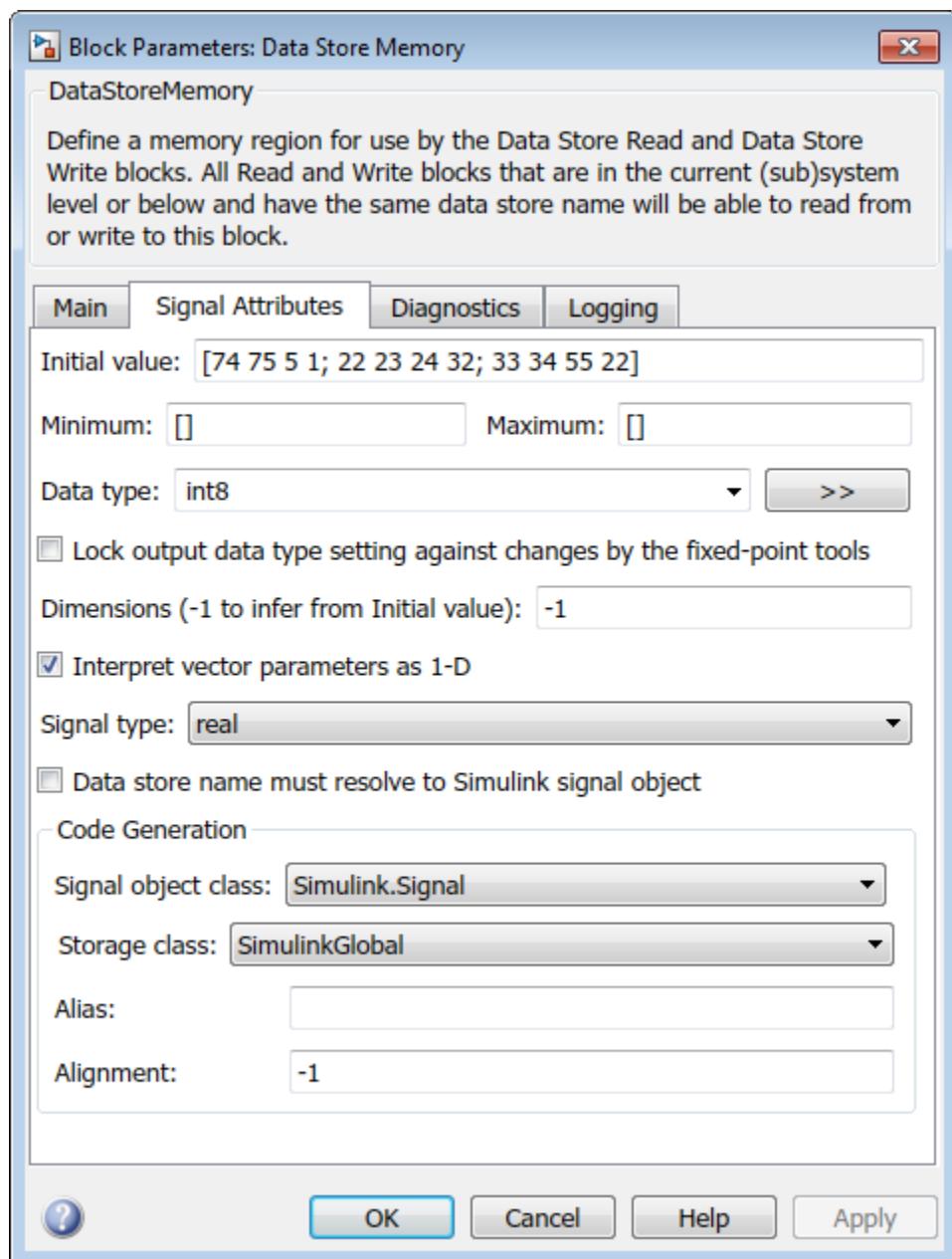
```
classdef GlobalSysObjMatrix < matlab.System
    methods (Access = protected)
        function y = stepImpl(obj)
            global B;
            B(1,:) = B(1,:)+1;
            y = B;
        end

        % Include getGlobalNamesImpl only if the class file is P-coded.
        function globalNames = getGlobalNamesImpl(~)
            globalNames = {"B"};
        end
    end
end
```

This model includes the `GlobalSysObjMatrix` object in a MATLAB System block and the associated Data Store Memory block.







See Also

Create Moving Average System object

This example shows how to create a System object™ that implements a moving average filter.

Introduction

System objects are MATLAB classes that derive from `matlab.System`. As a result, System objects all inherit a common public interface, which includes standard methods:

- `setup` — Initialize the object, typically at the beginning of a simulation
- `reset` — Clear the internal state of the object, bringing it back to its default post-initialization status
- `release` — Release any resources (memory, hardware, or OS-specific resources) used internally by the object

When you create new kinds of System objects, you provide specific implementations for all the preceding methods to determine its behavior.

In this example, you create and use the `movingAverageFilter` System object.

`movingAverageFilter` is a System object that computes the unweighted mean of the previous `WindowLength` input samples. `WindowLength` is the length of the moving average window. `movingAverageFilter` accepts single-precision and double-precision 2-D input matrices. Each column of the input matrix is treated as an independent (1-D) channel. The first dimension of the input defines the length of the channel (or the input frame size). `movingAverageFilter` independently computes the moving average of each input channel over time.

Create the System object File

In the MATLAB **Home** tab, create a new System object class by selecting **New > System Object > Basic**. The basic template for a System object opens in the MATLAB editor to guide you as you create the `movingAverageFilter` System object.

Rename the class `movingAverageFilter` and save the file as `movingAverageFilter.m`. To ensure you can use the System object, make sure you save the System object in a folder that is on the MATLAB path.

For your convenience, a complete `movingAverageFilter` System object file is available with this example. To open the completed class, enter:

```
edit movingAverageFilter.m
```

Add Properties

This System object needs four properties. First, add a public property `WindowLength` to control the length of the moving average window. Because the algorithm depends on this value being constant once data processing begins, the property is defined as nontunable. Additionally, the property only accepts real, positive integers. To ensure input satisfies these conditions, add property validators (see “Validate Property Values”). Set the default value of this property to 5.

```
properties(Nontunable)
    WindowLength (1,1){mustBeInteger,mustBePositive} = 5
end
```

Second, add two properties called `State` and `pNumChannels`. Users should not access either property, so use the `Access = private` attribute. `State` saves the state of the moving average

filter. `pNumChannels` stores the number of channels in your input. The value of this property is determined from the number of columns in the input.

```
properties(Access = private)
    State;
    pNumChannels = -1;
end
```

Finally, you need a property to store the FIR numerator coefficients. Add a property called `pCoefficients`. Because the coefficients do not change during data processing and you do not want users of the System object to access the coefficients, set the property attributes as `Nontunable`, `Access = private`.

```
properties(Access = private, Nontunable)
    pCoefficients
end
```

Add Constructor for Easy Creation

The System object constructor is a method that has the same name as the class (`movingAverageFilter` in this example). You implement a System object constructor to allow name-value pair inputs to the System object with the `setProperties` method. For example, with the constructor, users can use this syntax to create an instance of the System object: `filter = movingAverageFilter('WindowLength', 10)`. Do not use the constructor for anything else. All other setup tasks should be written in the `setupImpl` method.

```
methods
    function obj = movingAverageFilter(varargin)
        % Support name-value pair arguments when constructing object
        setProperties(obj,nargin varargin{:})
    end
end
```

Set Up and Initialization in `setupImpl`

The `setupImpl` method sets up the object and implements one-time initialization tasks. For this System object, modify the default `setupImpl` method to calculate the filter coefficients, the state, and the number of channels. The filter coefficients are computed based on the specified `WindowLength`. The filter state is initialized to zero. (Note that there are `WindowLength-1` states per input channel.) Finally, the number of channels is determined from the number of columns in the input.

For `setupImpl` and all `Impl` methods, you must set the method attribute `Access = protected` because users of the System object do not call these methods directly. Instead the back-end of the System object calls these methods through other user-facing functions.

```
function setupImpl(obj,x)
    % Perform one-time calculations, such as computing constants
    obj.pNumChannels = size(x,2);
    obj.pCoefficients = ones(1,obj.WindowLength)/obj.WindowLength;
    obj.State = zeros(obj.WindowLength-1,obj.pNumChannels,'like',x);
end
```

Define the Algorithm in stepImpl

The object's algorithm is defined in the `stepImpl` method. The algorithm in `stepImpl` is executed when the user of the System object calls the object at the command line. In this example, modify `stepImpl` to calculate the output and update the object's state values using the `filter` function.

```
function y = stepImpl(obj,u)
    [y,obj.State] = filter(obj.pCoefficients,1,u,obj.State);
end
```

Reset and Release

The state reset equations are defined in the `resetImpl` method. In this example, reset states to zero. Additionally, you need to add a `releaseImpl` method. From the **Editor** toolbar, select **Insert Method > Release resources**. The `releaseImpl` method is added to your System object. Modify `releaseImpl` to set the number of channels to `-1`, which allows new input to be used with the filter.

```
function resetImpl(obj)
    % Initialize / reset discrete-state properties
    obj.State(:) = 0;
end
function releaseImpl(obj)
    obj.pNumChannels = -1;
end
```

Validate Input

To validate inputs to your System object, you must implement a `validateInputsImpl` method. This method validates inputs at initialization and at each subsequent call where the input attributes (such as dimensions, data type, or complexity) change. From the toolbar, select **Insert Method > Validate inputs**. In the newly inserted `validateInputsImpl` method, call `validateattributes` to ensure that the input is a 2-D matrix with floating-point data.

```
function validateInputsImpl(~, u)
    validateattributes(u,{ 'double', 'single' }, { '2d', ...
        'nonsparse' }, ' ', 'input');
end
```

Object Saving and Loading

When you save an instance of a System object, `saveObjectImpl` defines what property and state values are saved in a MAT-file. If you do not define a `saveObjectImpl` method for your System object class, only public properties and properties with the `DiscreteState` attribute are saved. Select **Insert Method > Save in MAT-file**. Modify `saveObjectImpl` so that if the object is locked, the coefficients and number of channels is saved.

```
function s = saveObjectImpl(obj)
    s = saveObjectImpl@matlab.System(obj);
    if isLocked(obj)
        s.pCoefficients = obj.pCoefficients;
        s.pNumChannels = obj.pNumChannels;
    end
end
```

`loadObjectImpl` is the companion to `saveObjectImpl` because it defines how a saved object loads. When you load the saved object, the object loads in the same locked state. Select **Insert**

Method > Load from MAT-file. Modify `loadObjectImpl` so that if the object is locked, the coefficients and number of channels are loaded.

```
function loadObjectImpl(obj,s,wasLocked)
    if wasLocked
        obj.pCoefficients = s.pCoefficients;
        obj.pNumChannels = s.pNumChannels;
    end
    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
```

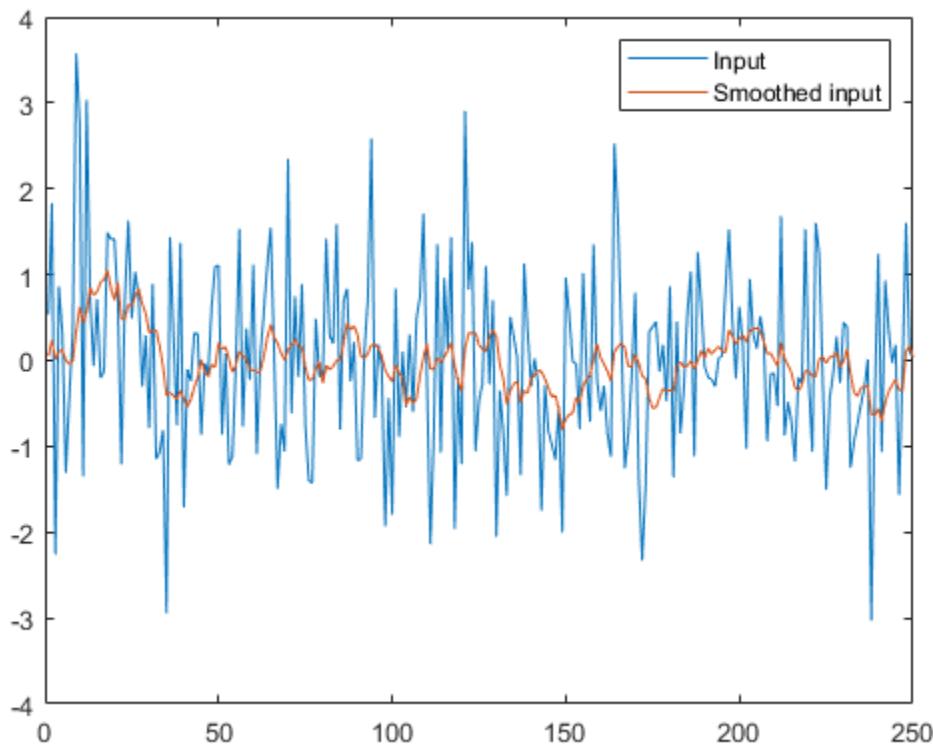
Use `movingAverageFilter` in MATLAB

Now that you have defined the System object, you can use the object in MATLAB. For example, use `movingAverageFilter` to remove noise from a noisy pulse sequence.

```
movingAverageFilter = movingAverageFilter('WindowLength',10);

t = (1:250)';
signal = randn(250,1);
smoothedSignal = movingAverageFilter(signal);

plot(t,signal,t,smoothedSignal);
legend(["Input","Smoothed input"])
```



Extend Your System object for Simulink

To use your System object in Simulink, see “Create Moving Average Filter Block with System Object” (Simulink).

Create New System Objects for File Input and Output

This example shows how to create and use two different System objects to facilitate the streaming of data in and out of MATLAB®: `TextFileReader` and `TextWriter`.

The objects discussed in this example address a number of realistic use cases, and they can be customized to achieve more advanced and specialized tasks.

Introduction

System objects are MATLAB classes that derive from `matlab.System`. As a result, System objects all inherit a common public interface, which includes standard methods:

- `setup` — Initialize the object, typically at the beginning of a simulation
- `reset` — Clear the internal state of the object, bringing it back to its default post-initialization status
- `release` — Release any resources (memory, hardware, or OS-specific resources) used internally by the object

When you create new kinds of System objects, you provide specific implementations for all the preceding methods to determine its behavior.

In this example we discuss the internal structure and the use of the following two System objects:

- `TextFileReader`
- `TextWriter`

To create these System objects for streaming data in and out of MATLAB, this example uses standard low-level file I/O functions available in MATLAB (like `fscanf`, `fread`, `fprintf`, and `fwrite`). By abstracting away most usage details of those functions, they aim to make the task of reading and writing streamed data simpler and more efficient.

This example includes the use of a number of advanced constructs to author System objects. For a more basic introduction to authoring System objects, see “Create System Objects”.

Definition of the Class `TextFileReader`

The `TextFileReader` class includes a class definition, public and private properties, a constructor, protected methods overridden from the `matlab.System` base class, and private methods. The `TextWriter` class is similarly structured.

Class Definition

The class definition states that the `TextFileReader` class is derived from both `matlab.System` and `matlab.system.mixin.FiniteSource`.

- ```
classdef (StrictDefaults)TextFileReader < matlab.System & matlab.system.mixin.FiniteSource
```
- `matlab.System` is required and is the base class for all System objects
  - `matlab.system.mixin.FiniteSource` indicates this class is a signal source with a finite number of data samples. For this type of class, in addition to the usual interface, the System object will also expose the `isDone` function. When `isDone` returns true, the object reached the end of the available data.

## Public Properties

Public properties can be changed by the user to adjust the behavior of the object to his or her particular application. `TextFileReader` has two nontunable public properties (they can only be changed before the first call to the object) and four tunable public properties. All the public properties have a default value. Default values are assigned to the corresponding properties when nothing else is specified by the user.

```
properties (Nontunable)
 Filename = 'tempfile.txt'
 HeaderLines = 4
end

properties
 DataFormat = '%g'
 Delimiter = ','
 SamplesPerFrame = 1024
 PlayCount = 1
end
```

## Private Properties

Private properties are not visible to the user and can serve a number of purposes, including

- To hold values computed only occasionally, then used with subsequent calls to the algorithm. For example, values used at initialization time, when `setup` is called or the object is called for the first time. This can save recomputing them at runtime and improve the performance of the core functionality
- To define the internal state of the object. For example, `pNumEofReached` stores the number of times that the end-of-file indicator was reached:

```
properties(Access = private)
 pFID = -1
 pNumChannels
 pLineFormat
 pNumEofReached = 0
end
```

## Constructor

The constructor is defined so that you can construct a `TextFileReader` object using name-value pairs. The constructor is called when a new instance of `TextDataReader` is created. The call to `setProperties` within the constructor allows setting properties with name-value pairs at construction. No other initialization tasks should be specified in the constructor. Instead, use the `setupImpl` method.

```
methods
 function obj = TextFileReader(varargin)
 setProperties(obj, nargin, varargin{:});
 end
end
```

## Overriding `matlab.System` Base Class Protected Methods

The public methods common to all System objects each have corresponding protected methods that they call internally. The names of these protected methods all include an `Impl` postfix. They can be implemented when defining the class to program the behavior of your System object.

For more information on the correspondence between the standard public methods and their internal implementations, please refer to “Summary of Call Sequence” on page 35-45.

For example, `TextFileReader` overrides these `Impl` methods:

- `setupImpl`
- `resetImpl`
- `stepImpl`
- `releaseImpl`
- `isDoneImpl`
- `processTunedPropertiesImpl`
- `loadObjectImpl`
- `saveObjectImpl`

### **Private Methods**

Private methods are only accessible from within other methods of the same class. They can be used to make the rest of the code more readable. They can also improve code reusability by grouping under separate routines code that is used multiple times in different parts of the class. For `TextFileReader`, private methods are created for:

- `getWorkingFID`
- `goToStartOfData`
- `peekCurrentLine`
- `lockNumberOfChannelsUsingCurrentLine`
- `readNDataRows`

### **Write and Read Data**

This example shows how you can use `TextFileReader` and `TextFileWriter` by:

- Creating a text file containing the samples of two different sinusoidal signals using `TextFileWriter`
- Read from the text file using `TextFileReader`.

### **Create a Simple Text File**

Create a new file to store two sinusoidal signals with frequencies of 50 Hz and 60 Hz. For each signal, the data stored is composed of 800 samples at a sampling rate of 8 kHz.

Create data samples:

```
fs = 8000;
tmax = 0.1;
t = (0:1/fs:tmax-1/fs)';
N = length(t);
f = [50,60];
data = sin(2*pi*t*f);
```

Form a header string to describe the data in a readable way for future use (optional step):

```
fileheader = sprintf(['The following contains %d samples of two ',...
 'sinusoids,\nwith frequencies %d Hz and %d Hz and a sample rate of',...
 '\n%d kHz\n\n'], N, f(1),f(2),fs/1000);
```

To store the signal to a text file, create a `TextWriter` object. The constructor of `TextWriter` needs the name of the target file and some optional parameters, which can be passed in as name-value pairs.

```
TxtWriter = TextWriter('Filename','sinewaves.txt','Header',fileheader)

TxtWriter =
 TextWriter with properties:

 Filename: 'sinewaves.txt'
 Header: 'The following contains 800 samples of two sinusoids,...'
 DataFormat: '%.18g'
 Delimiter: ','
```

`TextWriter` writes data to delimiter-separated ASCII files. Its public properties include:

- **Filename** — Name of the file to be written. If a file with this name already exists, it is overwritten. When operations start, the object begins writing to the file immediately following the header. The object then appends new data at each subsequent call to the object, until it is released. Calling `reset` resumes writing from the beginning of the file.
- **Header** — Character string, often composed of multiple lines and terminated by a newline character (`\n`). This is specified by the user and can be modified to embed human-readable information that describes the actual data.
- **DataFormat** — Format used to store each data sample. This can take any value assignable as Conversion Specifier within the `formatSpec` string used by the built-in MATLAB function `fprintf`. `DataFormat` applies to all channels written to the file. The default value for this property is '`'%.18g'`', which allows saving double precision floating point data in full precision.
- **Delimiter** — Character used to separate samples from different channels at the same time instant. Every line of the written file maps to a time instant, and it includes as many samples as the number of channels provided as input (in other words, the number of columns in the matrix input passed to the object).

To write all the available data to the file, a single call to `can` be used.

```
TxtWriter(data)
```

Release control of the file by calling the `release` function.

```
release(TxtWriter)
```

The data is now stored in the new file. To visually inspect the file, type:

```
edit('sinewaves.txt')
```

Because the header takes up three lines, the data starts on line 4.

In this simple case, the length of the whole signal is small, and it fits comfortably on system memory. Therefore, the data can be created all at once and written to a file in a single step.

There are cases when this approach is not possible or practical. For example, the data might be too large to fit into a single MATLAB variable (too large to fit on system memory). Alternatively, the data

might be created cyclically in a loop or streamed into MATLAB from an external source. In all these cases, streaming the data into the file can be done with an approach similar to the following example.

Use a streamed sine wave generator to create a frame of data per loop. Run the desired number of iterations to create the data and store it into the file:

```
frameLength = 32;
tmax = 10;
t = (0:1/fs:tmax-1/fs)';
N = length(t);
data = sin(2*pi*t*f);
numCycles = N/frameLength;

for k = 1:10 % Long running loop when you replace 10 with numCycles.
 dataFrame = sin(2*pi*t*f);
 TxtWriter(dataFrame)
end

release(TxtWriter)
```

## Read from Existing Text File

To read from the text file, create an instance of `TextFileReader`.

```
TxtReader = TextFileReader('Filename','sinewaves.txt','HeaderLines',3,'SamplesPerFrame',frameLength)

TxtReader =
TextFileReader with properties:

 Filename: 'sinewaves.txt'
 HeaderLines: 3
 DataFormat: '%g'
 Delimiter: ','
 SamplesPerFrame: 32
 PlayCount: 1
```

`TextFileReader` reads numeric data from delimiter-separated ASCII files. Its properties are similar to those of `TextWriter`. Some differences follow

- **HeaderLines** — Number of lines used by the header within the file specified in `Filename`. The first call to the object starts reading from line number `HeaderLines+1`. Subsequent calls to the object keep reading from the line immediately following the previously read line. Calling `reset` will resume reading from line `HeaderLines+1`.
- **Delimiter** — Character used to separate samples from different channels at the same time instant. In this case, the delimiter is also used to determine the number of data channels stored in the file. When the object is first run, the object counts the number of `Delimiter` characters at line `HeaderLines+1`, say `numDel`. Then for every time instant, the object reads `numChan = numDel+1` numeric values with format `DataFormat`. The matrix returned by the algorithm has size `SamplesPerFrame`-by-`numChan`.
- **SamplesPerFrame** — Number of lines read by each call to the object. This value is also the number of rows of the matrix returned as output. When the last available data rows are reached, there might be fewer than the required `SamplesPerFrame`. In that case, the available data are padded with zeros to obtain a matrix of size `SamplesPerFrame`-by-`numChan`. Once all the data are read, the algorithm simply returns `zeros(SamplesPerFrame,numChan)` until `reset` or `release` is called.

- **PlayCount** — Number of times the data in the file is read cyclically. If the object reaches the end of the file, and the file has not yet been read a number of times equal to **PlayCount**, reading resumes from the beginning of the data (line **HeaderLines+1**). If the last lines of the file do not provide enough samples to form a complete output matrix of size **SamplesPerFrame**-by-**numChan**, then the frame is completed using the initial data. Once the file is read **PlayCount** times, the output matrix returned by the algorithm is filled with zeros, and all calls to **isDone** return true unless **reset** or **release** is called. To loop through the available data indefinitely, **PlayCount** can be set to **Inf**.

To read the data from the text file, the more general streamed approach is used. This method of reading data is also relevant to dealing with very large data files. Preallocate a data frame with **frameLength** rows and 2 columns.

```
dataFrame = zeros(frameLength,2,'single');
```

Read from the text file and write to the binary file while data is present in the source text file. Notice how the method **isDone** is used to control the execution of the while loop.

```
while(~isDone(TxtReader))
 dataFrame(:) = TxtReader();
end

release(TxtReader)
```

## Summary

This example illustrated how to author and use System objects to read from and write to numeric data files. **TextFileReader** and **TextFileWriter** can be edited to perform special-purpose file reading and writing operations. You can also combine these custom System objects with built-in System objects such as **dsp.BinaryFileWriter** and **dsp.BinaryFileReader**.

For more information on authoring System objects for custom algorithms, see “Create System Objects”.

## See Also

### More About

- “Create Moving Average System object” on page 35-64
- “Tips for Defining System Objects” on page 35-50
- “Define Basic System Objects” on page 35-11

# Create Composite System object

This example shows how to create a System object™ composed of other System objects. This System object uses two moving average System objects to find the cross-correlation of two independent samples. The “Create Moving Average System object” on page 35-64 example explains in detail how to create a System object. This example focuses on how to use a System object within another System object.

## System objects as Private Properties

The ability to create more than one instance of a System object and having each instance manage its own state is one of the biggest advantages of using System objects over functions. The private properties `MovingAverageFilter1` and `MovingAverageFilter2` are used to store the two moving average filter objects.

```
properties (Access=private)
 % This example class contains two moving average filters (more can be added
 % in the same way)
 MovingAverageFilter1
 MovingAverageFilter2
end
```

## Set Up the Moving Average Filter

In the `setupImpl` method, create the two moving average System objects and initialize their public properties.

```
function setupImpl(obj,~)
 % Set up moving average objects with default values
 obj.MovingAverageFilter1 = movingAverageFilter('WindowLength',obj.WindowLength1);
 obj.MovingAverageFilter2 = movingAverageFilter('WindowLength',obj.WindowLength2);
end
```

## Work with Dependent Properties

The `WindowLength` public property from the `movingAverageFilter` System object is implemented as a *dependent* property in this example.

```
properties(Nontunable,Dependent)
 % WindowLength Moving window length
 WindowLength1;
 WindowLength2;
end
```

Whenever you assign a value to one of the dependent properties, the value is set in the corresponding moving average filter. When you read one of the dependent properties, the value is read from the corresponding moving average filter.

```
function set.WindowLength1(obj,WindowLength1)
 % Set the window length of one moving average filter
 obj.MovingAverageFilter1.WindowLength = WindowLength1;
end
function WindowLength = get.WindowLength1(obj)
 % Read window length from one of the moving average filters
 WindowLength = obj.MovingAverageFilter1.WindowLength;
end
function set.WindowLength2(obj,WindowLength2)
```

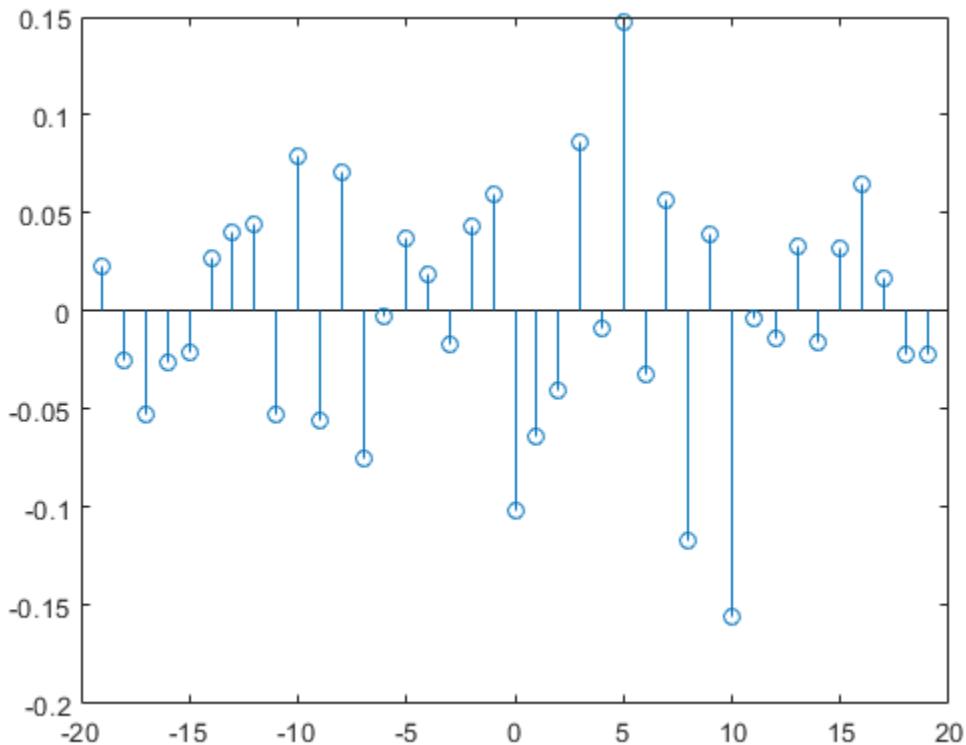
```
% Set the window length of one moving average filter
obj.MovingAverageFilter2.WindowLength = WindowLength2;
end
function WindowLength = get.WindowLength2(obj)
% Read window length from one of the moving average filters
WindowLength = obj.MovingAverageFilter2.WindowLength;
end
```

### Use the Cross-Correlation Object in MATLAB

Create random variables to calculate the cross-correlation of their moving averages, then view the results in a stem plot.

```
x = rand(20,1);
y = rand(20,1);
crossCorr = crossCorrelationMovingAverages('WindowLength1',1,'WindowLength2',5);

for iter = 1:100
 x = rand(20,1);
 y = rand(20,1);
 [corr,lags] = crossCorr(x,y);
 stem(lags,corr)
end
```



### More About

- “Create Moving Average System object” on page 35-64

- “Tips for Defining System Objects” on page 35-50

