How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Function Reference


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

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Format</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 1996</td>
<td>First printing</td>
<td>For MATLAB 5.0 (Release 8)</td>
</tr>
<tr>
<td>June 1997</td>
<td>Online only</td>
<td>Revised for MATLAB 5.1 (Release 9)</td>
</tr>
<tr>
<td>October 1997</td>
<td>Online only</td>
<td>Revised for MATLAB 5.2 (Release 10)</td>
</tr>
<tr>
<td>January 1999</td>
<td>Online only</td>
<td>Revised for MATLAB 5.3 (Release 11)</td>
</tr>
<tr>
<td>June 1999</td>
<td>Second printing</td>
<td>For MATLAB 5.3 (Release 11)</td>
</tr>
<tr>
<td>June 2001</td>
<td>Online only</td>
<td>Revised for MATLAB 6.1 (Release 12.1)</td>
</tr>
<tr>
<td>July 2002</td>
<td>Online only</td>
<td>Revised for 6.5 (Release 13)</td>
</tr>
<tr>
<td>June 2004</td>
<td>Online only</td>
<td>Revised for 7.0 (Release 14)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Online only</td>
<td>Revised for 7.3 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Revised for 7.4 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Online only</td>
<td>Revised for Version 7.5 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>Revised for Version 7.6 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for Version 7.7 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for Version 7.8 (Release 2009a)</td>
</tr>
</tbody>
</table>
# Function Reference

## Desktop Tools and Development Environment
- Startup and Shutdown ........................................ 1-3
- Command Window and History .............................. 1-4
- Help for Using MATLAB ..................................... 1-5
- Workspace, Search Path, and File Operations .......... 1-6
- Programming Tools ........................................ 1-8
- System ......................................................... 1-11

## Mathematics
- Arrays and Matrices ........................................ 1-14
- Linear Algebra ............................................. 1-20
- Elementary Math .......................................... 1-24
- Polynomials .................................................. 1-28
- Interpolation and Computational Geometry .......... 1-29
- Cartesian Coordinate System Conversion ........... 1-32
- Nonlinear Numerical Methods ............................ 1-33
- Specialized Math .......................................... 1-36
- Sparse Matrices ............................................ 1-37
- Math Constants ............................................ 1-41

## Data Analysis
- Basic Operations .......................................... 1-43
- Descriptive Statistics ..................................... 1-43
- Filtering and Convolution ................................ 1-44
- Interpolation and Regression ............................ 1-44
- Fourier Transforms ...................................... 1-45
- Derivatives and Integrals ................................ 1-45
- Time Series Objects ....................................... 1-46
- Time Series Collections ................................... 1-49

## Programming and Data Types
- Data Types ..................................................... 1-51
- Data Type Conversion ...................................... 1-59
- Operators and Special Characters ........................ 1-61
- Strings ......................................................... 1-64
Contents

Bit-Wise Operations ........................................ 1-67
Logical Operations ......................................... 1-67
Relational Operations ...................................... 1-68
Set Operations ............................................. 1-68
Date and Time Operations ................................ 1-69
Programming in MATLAB ................................... 1-69

Object-Oriented Programming .............................. 1-77
Classes and Objects ........................................ 1-77
Handle Classes ............................................. 1-78
Events and Listeners ...................................... 1-79
Meta-Classes ............................................... 1-79

File I/O .......................................................... 1-81
File Name Construction .................................... 1-81
File Opening, Loading, and Saving ....................... 1-82
Memory Mapping ............................................. 1-82
Low-Level File I/O ......................................... 1-82
Text Files ..................................................... 1-83
XML Documents ............................................ 1-84
Spreadsheets ............................................... 1-84
Scientific Data .............................................. 1-85
Audio and Audio/Video .................................... 1-88
Images ......................................................... 1-90
Internet Exchange .......................................... 1-91

Graphics ........................................................ 1-93
Basic Plots and Graphs .................................... 1-93
Plotting Tools ................................................. 1-94
Annotating Plots .......................................... 1-94
Specialized Plotting ....................................... 1-95
Bit-Mapped Images ........................................ 1-99
Printing ....................................................... 1-99
Handle Graphics ............................................. 1-100

3-D Visualization ............................................ 1-104
Surface and Mesh Plots .................................... 1-104
View Control .................................................. 1-106
Lighting ....................................................... 1-108
Transparency .................................................. 1-108
Volume Visualization ...................................... 1-108
Function Reference

Desktop Tools and Development Environment (p. 1-3)
Startup, Command Window, help, editing and debugging, tuning, other general functions

Mathematics (p. 1-13)
Arrays and matrices, linear algebra, other areas of mathematics

Data Analysis (p. 1-43)
Basic data operations, descriptive statistics, covariance and correlation, filtering and convolution, numerical derivatives and integrals, Fourier transforms, time series analysis

Programming and Data Types (p. 1-51)
Function/expression evaluation, program control, function handles, object oriented programming, error handling, operators, data types, dates and times, timers

Object-Oriented Programming (p. 1-77)
Functions for working with classes and objects

File I/O (p. 1-81)
General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images

Graphics (p. 1-93)
Line plots, annotating graphs, specialized plots, images, printing, Handle Graphics

3-D Visualization (p. 1-104)
Surface and mesh plots, view control, lighting and transparency, volume visualization
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI Development (p. 1-110)</td>
<td>GUIDE, programming graphical user interfaces</td>
</tr>
<tr>
<td>External Interfaces (p. 1-115)</td>
<td>Interfaces to shared libraries, Java, .NET, COM and ActiveX, Web services, and serial port devices, and C and Fortran routines</td>
</tr>
</tbody>
</table>
Desktop Tools and Development Environment

Start up and Shutdown (p. 1-3)
- Startup and shutdown options, preferences

Command Window and History (p. 1-4)
- Control Command Window and History, enter statements and run functions

Help for Using MATLAB (p. 1-5)
- Command line help, online documentation in the Help browser, demos

Workspace, Search Path, and File Operations (p. 1-6)
- Work with files, MATLAB search path, manage variables

Programming Tools (p. 1-8)
- Edit and debug M-files, improve performance, source control, publish results

System (p. 1-11)
- Identify current computer, license, product version, and more

Startup and Shutdown

- exit: Terminate MATLAB® program (same as quit)
- finish: Termination M-file for MATLAB program
- matlab (UNIX): Start MATLAB program (UNIX® platforms)
- matlab (Windows): Start MATLAB program (Windows® platforms)
- matlabrc: Startup M-file for MATLAB program
- prefdir: Directory containing preferences, history, and layout files
- preferences: Open Preferences dialog box
- quit: Terminate MATLAB program
Function Reference

startup

Startup M-file for user-defined options

userpath

View or change user portion of search path

Command Window and History

clc

Clear Command Window

commandhistory

Open Command History window, or select it if already open

commandwindow

Open Command Window, or select it if already open

diary

Save session to file

dos

Execute DOS command and return result

format

Set display format for output

home

Move cursor to upper-left corner of Command Window

matlabcolon (matlab:)

Run specified function via hyperlink

more

Control paged output for Command Window

perl

Call Perl script using appropriate operating system executable

system

Execute operating system command and return result

unix

Execute UNIX command and return result
## Help for Using MATLAB

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>builddocsearchdb</td>
<td>Build searchable documentation database</td>
</tr>
<tr>
<td>demo</td>
<td>Access product demos via Help browser</td>
</tr>
<tr>
<td>doc</td>
<td>Reference page in Help browser</td>
</tr>
<tr>
<td>docsearch</td>
<td>Open Help browser and search for specified term</td>
</tr>
<tr>
<td>echodemo</td>
<td>Run M-file demo step-by-step in Command Window</td>
</tr>
<tr>
<td>help</td>
<td>Help for functions in Command Window</td>
</tr>
<tr>
<td>helpbrowser</td>
<td>Open Help browser to access all online documentation and demos</td>
</tr>
<tr>
<td>helpwin</td>
<td>Provide access to M-file help for all functions</td>
</tr>
<tr>
<td>info</td>
<td>Information about contacting The MathWorks</td>
</tr>
<tr>
<td>lookfor</td>
<td>Search for keyword in all help entries</td>
</tr>
<tr>
<td>playshow</td>
<td>Run M-file demo (deprecated; use echodemo instead)</td>
</tr>
<tr>
<td>support</td>
<td>Open MathWorks Technical Support Web page</td>
</tr>
<tr>
<td>whatsnew</td>
<td>Release Notes for MathWorks™ products</td>
</tr>
</tbody>
</table>
## Workspace, Search Path, and File Operations

- **Workspace (p. 1-6)**
  - Manage variables

- **Search Path (p. 1-6)**
  - View and change MATLAB search path

- **File Operations (p. 1-7)**
  - View and change files and directories

### Workspace

- **assignin**
  - Assign value to variable in specified workspace

- **clear**
  - Remove items from workspace, freeing up system memory

- **evalin**
  - Execute MATLAB expression in specified workspace

- **exist**
  - Check existence of variable, function, directory, or Java™ programming language class

- **openvar**
  - Open workspace variable in Variable Editor or other graphical editing tool

- **pack**
  - Consolidate workspace memory

- **uiimport**
  - Open Import Wizard to import data

- **which**
  - Locate functions and files

- **who, whos**
  - List variables in workspace

- **workspace**
  - Open Workspace browser to manage workspace

### Search Path

- **addpath**
  - Add directories to search path

- **genpath**
  - Generate path string
partialpath          Partial pathname description
path                View or change search path
path2rc             Save current search path to pathdef.m file
pathsep             Path separator for current platform
pathtool            Open Set Path dialog box to view and change search path
restoredefaultpath  Restore default search path
rmpath              Remove directories from search path
savepath            Save current search path
userpath            View or change user portion of search path

**File Operations**
See also “File I/O” on page 1-81 functions.

  cd                  Change working directory
  copyfile           Copy file or directory
  delete             Remove files or graphics objects
  dir                 Directory listing
  exist               Check existence of variable, function, directory, or Java programming language class
  fileattrib         Set or get attributes of file or directory
  filebrowser        Open Current Directory browser, or select it if already open
  isdir               Determine whether input is directory
  lookfor            Search for keyword in all help entries
ls
matlabroot
mkdir
movefile
pwd
recycle
rehash
rmdir
toolboxdir
type
visdiff
what
which

Directory contents
Root directory
Make new directory
Move file or directory
Identify current directory
Set option to move deleted files to recycle folder
Refresh function and file system path caches
Remove directory
Root directory for specified toolbox
Display contents of file
Compare two text files, MAT-Files, or binary files
List MATLAB files in current directory
Locate functions and files

**Programming Tools**

M-File Editing and Debugging (p. 1-9)
M-File Performance (p. 1-9)
Source Control (p. 1-10)
Publishing (p. 1-10)

Edit and debug M-files
Improve performance and find potential problems in M-files
Interface MATLAB with source control system
Publish M-file code and results
**M-File Editing and Debugging**

- clipboard: Copy and paste strings to and from system clipboard
- datatipinfo: Produce short description of input variable
- dbclear: Clear breakpoints
- dbcont: Resume execution
- dbdown: Reverse workspace shift performed by dbup, while in debug mode
- dbquit: Quit debug mode
- dbstack: Function call stack
- dbstatus: List all breakpoints
- dbstep: Execute one or more lines from current breakpoint
- dbstop: Set breakpoints
- dbtype: List M-file with line numbers
- dbup: Shift current workspace to workspace of caller, while in debug mode
- debug: List M-file debugging functions
- edit: Edit or create M-file
- keyboard: Input from keyboard

**M-File Performance**

- bench: MATLAB benchmark
- mlint: Check M-files for possible problems
- mlintrpt: Run mlint for file or directory, reporting results in browser
- pack: Consolidate workspace memory
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile</td>
<td>Profile execution time for function</td>
</tr>
<tr>
<td>profsave</td>
<td>Save profile report in HTML format</td>
</tr>
<tr>
<td>rehash</td>
<td>Refresh function and file system path caches</td>
</tr>
<tr>
<td>sparse</td>
<td>Create sparse matrix</td>
</tr>
<tr>
<td>zeros</td>
<td>Create array of all zeros</td>
</tr>
</tbody>
</table>

**Source Control**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkin</td>
<td>Check files into source control system (UNIX platforms)</td>
</tr>
<tr>
<td>checkout</td>
<td>Check files out of source control system (UNIX platforms)</td>
</tr>
<tr>
<td>cmopts</td>
<td>Name of source control system</td>
</tr>
<tr>
<td>customverctrl</td>
<td>Allow custom source control system (UNIX platforms)</td>
</tr>
<tr>
<td>undocheckout</td>
<td>Undo previous checkout from source control system (UNIX platforms)</td>
</tr>
<tr>
<td>verctrl</td>
<td>Source control actions (Windows platforms)</td>
</tr>
</tbody>
</table>

**Publishing**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grabcode</td>
<td>MATLAB code from M-files published to HTML</td>
</tr>
<tr>
<td>notebook</td>
<td>Open M-book in Microsoft® Word software (on Microsoft Windows platforms)</td>
</tr>
<tr>
<td>publish</td>
<td>Publish M-file containing cells, saving output to file of specified type</td>
</tr>
<tr>
<td>snapnow</td>
<td>Force snapshot of image for inclusion in published document</td>
</tr>
</tbody>
</table>
### System

**Operating System Interface (p. 1-11)**

- **Exchange operating system information and commands with MATLAB**

**MATLAB Version and License (p. 1-12)**

- **Information about MATLAB version and license**

### Operating System Interface

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clipboard</td>
<td>Copy and paste strings to and from system clipboard</td>
</tr>
<tr>
<td>computer</td>
<td>Information about computer on which MATLAB software is running</td>
</tr>
<tr>
<td>dos</td>
<td>Execute DOS command and return result</td>
</tr>
<tr>
<td>getenv</td>
<td>Environment variable</td>
</tr>
<tr>
<td>hostid</td>
<td>Server host identification number</td>
</tr>
<tr>
<td>maxNumCompThreads</td>
<td>Control maximum number of computational threads</td>
</tr>
<tr>
<td>perl</td>
<td>Call Perl script using appropriate operating system executable</td>
</tr>
<tr>
<td>setenv</td>
<td>Set environment variable</td>
</tr>
<tr>
<td>system</td>
<td>Execute operating system command and return result</td>
</tr>
<tr>
<td>unix</td>
<td>Execute UNIX command and return result</td>
</tr>
<tr>
<td>winqueryreg</td>
<td>Item from Windows registry</td>
</tr>
</tbody>
</table>
### MATLAB Version and License

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ismac</td>
<td>Determine if version is for Mac OS® X platform</td>
</tr>
<tr>
<td>ispc</td>
<td>Determine if version is for Windows (PC) platform</td>
</tr>
<tr>
<td>isstudent</td>
<td>Determine if version is Student Version</td>
</tr>
<tr>
<td>isunix</td>
<td>Determine if version is for UNIX platform</td>
</tr>
<tr>
<td>javachk</td>
<td>Generate error message based on Sun™ Java feature support</td>
</tr>
<tr>
<td>license</td>
<td>Return license number or perform licensing task</td>
</tr>
<tr>
<td>prefdir</td>
<td>Directory containing preferences, history, and layout files</td>
</tr>
<tr>
<td>usejava</td>
<td>Determine whether Sun Java feature is supported in MATLAB software</td>
</tr>
<tr>
<td>ver</td>
<td>Version information for MathWorks products</td>
</tr>
<tr>
<td>verLessThan</td>
<td>Compare toolbox version to specified version string</td>
</tr>
<tr>
<td>version</td>
<td>Version number for MATLAB</td>
</tr>
</tbody>
</table>
# Mathematics

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays and Matrices (p. 1-14)</td>
<td>Basic array operators and operations, creation of elementary and specialized arrays and matrices</td>
</tr>
<tr>
<td>Linear Algebra (p. 1-20)</td>
<td>Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization</td>
</tr>
<tr>
<td>Elementary Math (p. 1-24)</td>
<td>Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math</td>
</tr>
<tr>
<td>Polynomials (p. 1-28)</td>
<td>Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion</td>
</tr>
<tr>
<td>Interpolation and Computational Geometry (p. 1-29)</td>
<td>Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation</td>
</tr>
<tr>
<td>Cartesian Coordinate System Conversion (p. 1-32)</td>
<td>Conversions between Cartesian and polar or spherical coordinates</td>
</tr>
<tr>
<td>Nonlinear Numerical Methods (p. 1-33)</td>
<td>Differential equations, optimization, integration</td>
</tr>
<tr>
<td>Specialized Math (p. 1-36)</td>
<td>Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions</td>
</tr>
<tr>
<td>Sparse Matrices (p. 1-37)</td>
<td>Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations</td>
</tr>
<tr>
<td>Math Constants (p. 1-41)</td>
<td>Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy</td>
</tr>
</tbody>
</table>
**Arrays and Matrices**

**Basic Information (p. 1-14)**
Display array contents, get array information, determine array type

**Operators (p. 1-15)**
Arithmetic operators

**Elementary Matrices and Arrays (p. 1-16)**
Create elementary arrays of different types, generate arrays for plotting, array indexing, etc.

**Array Operations (p. 1-17)**
Operate on array content, apply function to each array element, find cumulative product or sum, etc.

**Array Manipulation (p. 1-18)**
Create, sort, rotate, permute, reshape, and shift array contents

**Specialized Matrices (p. 1-19)**
Create Hadamard, Companion, Hankel, Vandermonde, Pascal matrices, etc.

---

**Basic Information**

disp
Display text or array

display
Display text or array (overloaded method)

isempty
Determine whether array is empty

isequal
Test arrays for equality

isequalwithequalnans
Test arrays for equality, treating NaNs as equal

isfinite
Array elements that are finite

isfloat
Determine whether input is floating-point array

isinf
Array elements that are infinite

isinteger
Determine whether input is integer array
islogical  Determine whether input is logical array
isnan     Array elements that are NaN
isnumeric Determine whether input is numeric array
isscalar  Determine whether input is scalar
issparse  Determine whether input is sparse
isvector  Determine whether input is vector
length    Length of vector
max       Largest elements in array
min       Smallest elements in array
ndims     Number of array dimensions
numel     Number of elements in array or subscripted array expression
size      Array dimensions

Operators

+         Addition
+-        Unary plus
-         Subtraction
--        Unary minus
*         Matrix multiplication
^         Matrix power
\         Backslash or left matrix divide
/         Slash or right matrix divide
\'        Transpose
\.'       Nonconjugated transpose
\.*       Array multiplication (element-wise)
.\^ Array power (element-wise)
.\ Left array divide (element-wise)
./ Right array divide (element-wise)

**Elementary Matrices and Arrays**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blkdiag</td>
<td>Construct block diagonal matrix from input arguments</td>
</tr>
<tr>
<td>create (RandStream)</td>
<td>Create random number streams</td>
</tr>
<tr>
<td>diag</td>
<td>Diagonal matrices and diagonals of matrix</td>
</tr>
<tr>
<td>eye</td>
<td>Identity matrix</td>
</tr>
<tr>
<td>freqspace</td>
<td>Frequency spacing for frequency response</td>
</tr>
<tr>
<td>get (RandStream)</td>
<td>Random stream properties</td>
</tr>
<tr>
<td>getDefaultStream (RandStream)</td>
<td>Default random number stream</td>
</tr>
<tr>
<td>ind2sub</td>
<td>Subscripts from linear index</td>
</tr>
<tr>
<td>linspace</td>
<td>Generate linearly spaced vectors</td>
</tr>
<tr>
<td>list (RandStream)</td>
<td>Random number generator algorithms</td>
</tr>
<tr>
<td>logspace</td>
<td>Generate logarithmically spaced vectors</td>
</tr>
<tr>
<td>meshgrid</td>
<td>Generate X and Y arrays for 3-D plots</td>
</tr>
<tr>
<td>ndgrid</td>
<td>Generate arrays for N-D functions and interpolation</td>
</tr>
<tr>
<td>ones</td>
<td>Create array of all ones</td>
</tr>
<tr>
<td>rand</td>
<td>Uniformly distributed pseudorandom numbers</td>
</tr>
<tr>
<td>rand (RandStream)</td>
<td>Uniformly distributed random numbers</td>
</tr>
</tbody>
</table>
randi
Uniformly distributed pseudorandom integers
randi (RandStream)
Uniformly distributed pseudorandom integers
randn
Normally distributed pseudorandom numbers
randn (RandStream)
Normally distributed pseudorandom numbers
randperm (RandStream)
RandStream Random number stream
RandStream (RandStream)
RandStream Random number stream
set (RandStream)
Set random stream property
setDefaultStream (RandStream)
Set default random number stream
sub2ind
Single index from subscripts
zeros
Create array of all zeros

Array Operations

See “Linear Algebra” on page 1-20 and “Elementary Math” on page 1-24 for other array operations.

accumarray
Construct array with accumulation
arrayfun
Apply function to each element of array
bsxfun
Apply element-by-element binary operation to two arrays with singleton expansion enabled
cast
Cast variable to different data type
cross
Vector cross product
cumprod
Cumulative product
cumsum
Cumulative sum
### Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot</td>
<td>Vector dot product</td>
</tr>
<tr>
<td>idivide</td>
<td>Integer division with rounding option</td>
</tr>
<tr>
<td>kron</td>
<td>Kronecker tensor product</td>
</tr>
<tr>
<td>prod</td>
<td>Product of array elements</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of array elements</td>
</tr>
<tr>
<td>tril</td>
<td>Lower triangular part of matrix</td>
</tr>
<tr>
<td>triu</td>
<td>Upper triangular part of matrix</td>
</tr>
</tbody>
</table>

### Array Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blkdiag</td>
<td>Construct block diagonal matrix from input arguments</td>
</tr>
<tr>
<td>cat</td>
<td>Concatenate arrays along specified dimension</td>
</tr>
<tr>
<td>circshift</td>
<td>Shift array circularly</td>
</tr>
<tr>
<td>diag</td>
<td>Diagonal matrices and diagonals of matrix</td>
</tr>
<tr>
<td>end</td>
<td>Terminate block of code, or indicate last array index</td>
</tr>
<tr>
<td>flipdim</td>
<td>Flip array along specified dimension</td>
</tr>
<tr>
<td>flipl</td>
<td>Flip matrix left to right</td>
</tr>
<tr>
<td>flipu</td>
<td>Flip matrix up to down</td>
</tr>
<tr>
<td>horzcat</td>
<td>Concatenate arrays horizontally</td>
</tr>
<tr>
<td>inline</td>
<td>Construct inline object</td>
</tr>
<tr>
<td>ipermute</td>
<td>Inverse permute dimensions of N-D array</td>
</tr>
<tr>
<td>permute</td>
<td>Rearrange dimensions of N-D array</td>
</tr>
<tr>
<td>repmat</td>
<td>Replicate and tile array</td>
</tr>
<tr>
<td>reshape</td>
<td>Reshape array</td>
</tr>
</tbody>
</table>
rot90
shiftdim
sort
sortrows
squeeze
vectorize
vertcat

Rotate matrix 90 degrees
Shift dimensions
Sort array elements in ascending or descending order
Sort rows in ascending order
Remove singleton dimensions
Vectorize expression
Concatenate arrays vertically

Specialized Matrices

compan
gallery
hadamard
hankel
hilb
invhilb
magic
pascal
rosser
toeplitz
vander
wilkinson

Companion matrix
Test matrices
Hadamard matrix
Hankel matrix
Hilbert matrix
Inverse of Hilbert matrix
Magic square
Pascal matrix
Classic symmetric eigenvalue test problem
Toeplitz matrix
Vandermonde matrix
Wilkinson’s eigenvalue test matrix
**Linear Algebra**

Matrix Analysis (p. 1-20)  
Compute norm, rank, determinant, condition number, etc.

Linear Equations (p. 1-21)  
Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.

Eigenvalues and Singular Values (p. 1-22)  
Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.

Matrix Logarithms and Exponentials (p. 1-23)  
Matrix logarithms, exponentials, square root

Factorization (p. 1-23)  
Cholesky, LU, and QR factorizations, diagonal forms, singular value decomposition

**Matrix Analysis**

cond  
Condition number with respect to inversion

condeig  
Condition number with respect to eigenvalues

det  
Matrix determinant

norm  
Vector and matrix norms

normest  
2-norm estimate

null  
Null space

orth  
Range space of matrix

rank  
Rank of matrix

rcond  
Matrix reciprocal condition number estimate

rref  
Reduced row echelon form
subspace Angle between two subspaces
trace Sum of diagonal elements

**Linear Equations**

chol Cholesky factorization
cholinc Sparse incomplete Cholesky and Cholesky-Infinity factorizations
cond Condition number with respect to inversion
condest 1-norm condition number estimate
funm Evaluate general matrix function
ilu Sparse incomplete LU factorization
inv Matrix inverse
linsolve Solve linear system of equations
lscov Least-squares solution in presence of known covariance
lsqnonneg Solve nonnegative least-squares constraints problem
lu LU matrix factorization
luinc Sparse incomplete LU factorization
pinv Moore-Penrose pseudoinverse of matrix
qr Orthogonal-triangular decomposition
rcond Matrix reciprocal condition number estimate
Eigenvalues and Singular Values

- balance: Diagonal scaling to improve eigenvalue accuracy
- cdf2rdf: Convert complex diagonal form to real block diagonal form
- condeig: Condition number with respect to eigenvalues
- eig: Eigenvalues and eigenvectors
- eigs: Largest eigenvalues and eigenvectors of matrix
- gsvd: Generalized singular value decomposition
- hess: Hessenberg form of matrix
- ordeig: Eigenvalues of quasitriangular matrices
- ordqz: Reorder eigenvalues in QZ factorization
- ordschur: Reorder eigenvalues in Schur factorization
- poly: Polynomial with specified roots
- polyeig: Polynomial eigenvalue problem
- rsf2csf: Convert real Schur form to complex Schur form
- schur: Schur decomposition
- sqrtm: Matrix square root
- ss2tf: Convert state-space filter parameters to transfer function form
- svd: Singular value decomposition
- svds: Find singular values and vectors
Matrix Logarithms and Exponentials

- expm: Matrix exponential
- logm: Matrix logarithm
- sqrtm: Matrix square root

Factorization

- balance: Diagonal scaling to improve eigenvalue accuracy
- cdf2rdf: Convert complex diagonal form to real block diagonal form
- chol: Cholesky factorization
- cholinc: Sparse incomplete Cholesky and Cholesky-Infinity factorizations
- cholupdate: Rank 1 update to Cholesky factorization
- gsvd: Generalized singular value decomposition
- ilu: Sparse incomplete LU factorization
- lu: LU matrix factorization
- luinc: Sparse incomplete LU factorization
- planerot: Givens plane rotation
- qr: Orthogonal-triangular decomposition
- qrdelete: Remove column or row from QR factorization
- qrinsert: Insert column or row into QR factorization
- qrupdate
qz
QZ factorization for generalized eigenvalues

rsf2csf
Convert real Schur form to complex Schur form

svd
Singular value decomposition

**Elementary Math**

**Trigonometric (p. 1-24)**
Trigonometric functions with results in radians or degrees

**Exponential (p. 1-26)**
Exponential, logarithm, power, and root functions

**Complex (p. 1-26)**
Numbers with real and imaginary components, phase angles

**Rounding and Remainder (p. 1-27)**
Rounding, modulus, and remainder

**Discrete Math (p. 1-27)**
Prime factors, factorials, permutations, rational fractions, least common multiple, greatest common divisor

**Trigonometric**

acos
Inverse cosine; result in radians

acosd
Inverse cosine; result in degrees

acosh
Inverse hyperbolic cosine

acot
Inverse cotangent; result in radians

acotd
Inverse cotangent; result in degrees

acoth
Inverse hyperbolic cotangent

acsc
Inverse cosecant; result in radians

acscd
Inverse cosecant; result in degrees

acsch
Inverse hyperbolic cosecant
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asec</td>
<td>Inverse secant; result in radians</td>
</tr>
<tr>
<td>asecd</td>
<td>Inverse secant; result in degrees</td>
</tr>
<tr>
<td>asech</td>
<td>Inverse hyperbolic secant</td>
</tr>
<tr>
<td>asin</td>
<td>Inverse sine; result in radians</td>
</tr>
<tr>
<td>asind</td>
<td>Inverse sine; result in degrees</td>
</tr>
<tr>
<td>asinh</td>
<td>Inverse hyperbolic sine</td>
</tr>
<tr>
<td>atan</td>
<td>Inverse tangent; result in radians</td>
</tr>
<tr>
<td>atan2</td>
<td>Four-quadrant inverse tangent</td>
</tr>
<tr>
<td>atand</td>
<td>Inverse tangent; result in degrees</td>
</tr>
<tr>
<td>atanh</td>
<td>Inverse hyperbolic tangent</td>
</tr>
<tr>
<td>cos</td>
<td>Cosine of argument in radians</td>
</tr>
<tr>
<td>cosd</td>
<td>Cosine of argument in degrees</td>
</tr>
<tr>
<td>cosh</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>cot</td>
<td>Cotangent of argument in radians</td>
</tr>
<tr>
<td>cotd</td>
<td>Cotangent of argument in degrees</td>
</tr>
<tr>
<td>coth</td>
<td>Hyperbolic cotangent</td>
</tr>
<tr>
<td>csc</td>
<td>Cosecant of argument in radians</td>
</tr>
<tr>
<td>csdcd</td>
<td>Cosecant of argument in degrees</td>
</tr>
<tr>
<td>csch</td>
<td>Hyperbolic cosecant</td>
</tr>
<tr>
<td>hypot</td>
<td>Square root of sum of squares</td>
</tr>
<tr>
<td>sec</td>
<td>Secant of argument in radians</td>
</tr>
<tr>
<td>secd</td>
<td>Secant of argument in degrees</td>
</tr>
<tr>
<td>sech</td>
<td>Hyperbolic secant</td>
</tr>
<tr>
<td>sin</td>
<td>Sine of argument in radians</td>
</tr>
<tr>
<td>sind</td>
<td>Sine of argument in degrees</td>
</tr>
<tr>
<td>sinh</td>
<td>Hyperbolic sine of argument in radians</td>
</tr>
<tr>
<td>tan</td>
<td>Tangent of argument in radians</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>tand</code></td>
<td>Tangent of argument in degrees</td>
</tr>
<tr>
<td><code>tanh</code></td>
<td>Hyperbolic tangent</td>
</tr>
<tr>
<td><strong>Exponential</strong></td>
<td></td>
</tr>
<tr>
<td><code>exp</code></td>
<td>Exponential</td>
</tr>
<tr>
<td><code>expm1</code></td>
<td>Compute $\exp(x) - 1$ accurately for small values of $x$</td>
</tr>
<tr>
<td><code>log</code></td>
<td>Natural logarithm</td>
</tr>
<tr>
<td><code>log10</code></td>
<td>Common (base 10) logarithm</td>
</tr>
<tr>
<td><code>log1p</code></td>
<td>Compute $\log(1+x)$ accurately for small values of $x$</td>
</tr>
<tr>
<td><code>log2</code></td>
<td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td>
</tr>
<tr>
<td><code>nextpow2</code></td>
<td>Next higher power of 2</td>
</tr>
<tr>
<td><code>nthroot</code></td>
<td>Real $n$th root of real numbers</td>
</tr>
<tr>
<td><code>pow2</code></td>
<td>Base 2 power and scale floating-point numbers</td>
</tr>
<tr>
<td><code>reallog</code></td>
<td>Natural logarithm for nonnegative real arrays</td>
</tr>
<tr>
<td><code>realpow</code></td>
<td>Array power for real-only output</td>
</tr>
<tr>
<td><code>realsqrt</code></td>
<td>Square root for nonnegative real arrays</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>Square root</td>
</tr>
<tr>
<td><strong>Complex</strong></td>
<td></td>
</tr>
<tr>
<td><code>abs</code></td>
<td>Absolute value and complex magnitude</td>
</tr>
<tr>
<td><code>angle</code></td>
<td>Phase angle</td>
</tr>
</tbody>
</table>
complex

complex Construct complex data from real and imaginary components

conj

Complex conjugate

cplxpair

Sort complex numbers into complex conjugate pairs

i

Imaginary unit

imag

Imaginary part of complex number

isreal

Check if input is real array

j

Imaginary unit

real

Real part of complex number

sign

Signum function

unwrap

Correct phase angles to produce smoother phase plots

Rounding and Remainder

ceil

Round toward positive infinity

fix

Round toward zero

floor

Round toward negative infinity

idivide

Integer division with rounding option

mod

Modulus after division

rem

Remainder after division

round

Round to nearest integer

Discrete Math

factor

Prime factors

factorial

Factorial function

gcd

Greatest common divisor
### Polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv</td>
<td>Convolution and polynomial multiplication</td>
</tr>
<tr>
<td>deconv</td>
<td>Deconvolution and polynomial division</td>
</tr>
<tr>
<td>poly</td>
<td>Polynomial with specified roots</td>
</tr>
<tr>
<td>polyder</td>
<td>Polynomial derivative</td>
</tr>
<tr>
<td>polyeig</td>
<td>Polynomial eigenvalue problem</td>
</tr>
<tr>
<td>polyfit</td>
<td>Polynomial curve fitting</td>
</tr>
<tr>
<td>polyint</td>
<td>Integrate polynomial analytically</td>
</tr>
<tr>
<td>polyval</td>
<td>Polynomial evaluation</td>
</tr>
<tr>
<td>polyvalm</td>
<td>Matrix polynomial evaluation</td>
</tr>
<tr>
<td>residue</td>
<td>Convert between partial fraction expansion and polynomial coefficients</td>
</tr>
<tr>
<td>roots</td>
<td>Polynomial roots</td>
</tr>
</tbody>
</table>
Interpolation and Computational Geometry

Interpolation (p. 1-29)
Data interpolation, data gridding, polynomial evaluation, nearest point search

Delaunay Triangulation and Tessellation (p. 1-30)
Delaunay triangulation and tessellation, triangular surface and mesh plots

Convex Hull (p. 1-32)
Plot convex hull, plotting functions

Voronoi Diagrams (p. 1-32)
Plot Voronoi diagram, patch graphics object, plotting functions

Domain Generation (p. 1-32)
Generate arrays for 3-D plots, or for N-D functions and interpolation

Interpolation

dsearch
Search Delaunay triangulation for nearest point

dsearchn
N-D nearest point search

griddata
Data gridding

griddata3
Data gridding and hypersurface fitting for 3-D data

griddatatan
Data gridding and hypersurface fitting (dimension >= 2)

interp1
1-D data interpolation (table lookup)

interp1q
Quick 1-D linear interpolation

interp2
2-D data interpolation (table lookup)

interp3
3-D data interpolation (table lookup)

interpft
1-D interpolation using FFT method

interpn
N-D data interpolation (table lookup)

meshgrid
Generate X and Y arrays for 3-D plots
### Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkpp</td>
<td>Make piecewise polynomial</td>
</tr>
<tr>
<td>ndgrid</td>
<td>Generate arrays for N-D functions and interpolation</td>
</tr>
<tr>
<td>padecoef</td>
<td>Padé approximation of time delays</td>
</tr>
<tr>
<td>pchip</td>
<td>Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)</td>
</tr>
<tr>
<td>ppval</td>
<td>Evaluate piecewise polynomial</td>
</tr>
<tr>
<td>spline</td>
<td>Cubic spline data interpolation</td>
</tr>
<tr>
<td>tsearchn</td>
<td>N-D closest simplex search</td>
</tr>
<tr>
<td>unmkpp</td>
<td>Piecewise polynomial details</td>
</tr>
</tbody>
</table>

### Delaunay Triangulation and Tessellation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baryToCart (TriRep)</td>
<td>Converts point coordinates from barycentric to Cartesian</td>
</tr>
<tr>
<td>cartToBary (TriRep)</td>
<td>Convert point coordinates from cartesian to barycentric</td>
</tr>
<tr>
<td>circumcenters (TriRep)</td>
<td>Circumcenters of specified simplices</td>
</tr>
<tr>
<td>convexHull (DelaunayTri)</td>
<td>Convex hull</td>
</tr>
<tr>
<td>delaunay</td>
<td>Delaunay triangulation</td>
</tr>
<tr>
<td>delaunay3</td>
<td>3-D Delaunay tessellation</td>
</tr>
<tr>
<td>delaunayn</td>
<td>N-D Delaunay tessellation</td>
</tr>
<tr>
<td>DelaunayTri</td>
<td>Construct Delaunay triangulation</td>
</tr>
<tr>
<td>DelaunayTri</td>
<td>Delaunay triangulation in 2-D and 3-D</td>
</tr>
<tr>
<td>dsearch</td>
<td>Search Delaunay triangulation for nearest point</td>
</tr>
<tr>
<td>dsearchn</td>
<td>N-D nearest point search</td>
</tr>
<tr>
<td>edgeAttachments (TriRep)</td>
<td>Simplices attached to specified edges</td>
</tr>
<tr>
<td>edges (TriRep)</td>
<td>Triangulation edges</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>faceNormals (TriRep)</td>
<td>Unit normals to specified triangles</td>
</tr>
<tr>
<td>featureEdges (TriRep)</td>
<td>Sharp edges of surface triangulation</td>
</tr>
<tr>
<td>freeBoundary (TriRep)</td>
<td>Facets referenced by only one simplex</td>
</tr>
<tr>
<td>incenters (TriRep)</td>
<td>Incenters of specified simplices</td>
</tr>
<tr>
<td>inOutStatus (DelaunayTri)</td>
<td>Status of triangles in 2-D constrained Delaunay triangulation</td>
</tr>
<tr>
<td>isEdge (TriRep)</td>
<td>Test if vertices are joined by edge</td>
</tr>
<tr>
<td>nearestNeighbor (DelaunayTri)</td>
<td>Point closest to specified location</td>
</tr>
<tr>
<td>neighbors (TriRep)</td>
<td>Simplex neighbor information</td>
</tr>
<tr>
<td>pointLocation (DelaunayTri)</td>
<td>Simplex containing specified location</td>
</tr>
<tr>
<td>size (TriRep)</td>
<td>Size of triangulation matrix</td>
</tr>
<tr>
<td>tetramesh</td>
<td>Tetrahedron mesh plot</td>
</tr>
<tr>
<td>trimesh</td>
<td>Triangular mesh plot</td>
</tr>
<tr>
<td>triplot</td>
<td>2-D triangular plot</td>
</tr>
<tr>
<td>TriRep</td>
<td>Triangulation representation</td>
</tr>
<tr>
<td>TriRep</td>
<td>Triangulation representation</td>
</tr>
<tr>
<td>TriScatteredInterp</td>
<td>Interpolate scattered data</td>
</tr>
<tr>
<td>TriScatteredInterp</td>
<td>Triangular surface plot</td>
</tr>
<tr>
<td>trisurf</td>
<td>Search for enclosing Delaunay triangle</td>
</tr>
<tr>
<td>tsearch</td>
<td>N-D closest simplex search</td>
</tr>
<tr>
<td>tsearchn</td>
<td>Return simplices attached to specified vertices</td>
</tr>
<tr>
<td>vertexAttachments (TriRep)</td>
<td>Voronoi diagram</td>
</tr>
<tr>
<td>voronoiDiagram (DelaunayTri)</td>
<td></td>
</tr>
</tbody>
</table>
### Convex Hull

- `convhull`: Convex hull
- `convhulln`: N-D convex hull
- `patch`: Create patch graphics object
- `plot`: 2-D line plot
- `trisurf`: Triangular surface plot

### Voronoi Diagrams

- `dsearch`: Search Delaunay triangulation for nearest point
- `patch`: Create patch graphics object
- `plot`: 2-D line plot
- `voronoi`: Voronoi diagram
- `voronoin`: N-D Voronoi diagram

### Domain Generation

- `meshgrid`: Generate X and Y arrays for 3-D plots
- `ndgrid`: Generate arrays for N-D functions and interpolation

### Cartesian Coordinate System Conversion

- `cart2pol`: Transform Cartesian coordinates to polar or cylindrical
- `cart2sph`: Transform Cartesian coordinates to spherical
pol2cart  Transform polar or cylindrical coordinates to Cartesian
sph2cart  Transform spherical coordinates to Cartesian

**Nonlinear Numerical Methods**

Ordinary Differential Equations  (p. 1-33)  Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution
Delay Differential Equations  (p. 1-34)  Solve delay differential equations with constant and general delays, set solver options, evaluate solution
Partial Differential Equations  (p. 1-35)  Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution
Optimization (p. 1-35)  Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem
Numerical Integration (Quadrature)  (p. 1-36)  Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

**Ordinary Differential Equations**

decic  Compute consistent initial conditions for ode15i
deval  Evaluate solution of differential equation problem
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode15i</td>
<td>Solve fully implicit differential equations, variable order method</td>
</tr>
<tr>
<td>ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb</td>
<td>Solve initial value problems for ordinary differential equations</td>
</tr>
<tr>
<td>odefile</td>
<td>Define differential equation problem for ordinary differential equation solvers</td>
</tr>
<tr>
<td>odeget</td>
<td>Ordinary differential equation options parameters</td>
</tr>
<tr>
<td>odeset</td>
<td>Create or alter options structure for ordinary differential equation solvers</td>
</tr>
<tr>
<td>odextend</td>
<td>Extend solution of initial value problem for ordinary differential equation</td>
</tr>
</tbody>
</table>

**Delay Differential Equations**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dde23</td>
<td>Solve delay differential equations (DDEs) with constant delays</td>
</tr>
<tr>
<td>ddeget</td>
<td>Extract properties from delay differential equations options structure</td>
</tr>
<tr>
<td>ddesd</td>
<td>Solve delay differential equations (DDEs) with general delays</td>
</tr>
<tr>
<td>ddeset</td>
<td>Create or alter delay differential equations options structure</td>
</tr>
<tr>
<td>deval</td>
<td>Evaluate solution of differential equation problem</td>
</tr>
</tbody>
</table>
**Boundary Value Problems**

- **bvp4c**: Solve boundary value problems for ordinary differential equations
- **bvp5c**: Solve boundary value problems for ordinary differential equations
- **bvpget**: Extract properties from options structure created with `bvpset`
- **bvpinit**: Form initial guess for `bvp4c`
- **bvpset**: Create or alter options structure of boundary value problem
- **bvpxtend**: Form guess structure for extending boundary value solutions
- **deval**: Evaluate solution of differential equation problem

**Partial Differential Equations**

- **pdepe**: Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D
- **pdeval**: Evaluate numerical solution of PDE using output of `pdepe`

**Optimization**

- **fminbnd**: Find minimum of single-variable function on fixed interval
- **fminsearch**: Find minimum of unconstrained multivariable function using derivative-free method
- **fzero**: Find root of continuous function of one variable
lsqnonneg Solve nonnegative least-squares constraints problem
optimget Optimization options values
optimset Create or edit optimization options structure

**Numerical Integration (Quadrature)**

dblquad Numerically evaluate double integral over a rectangle
quad Numerically evaluate integral, adaptive Simpson quadrature
quad2d Numerically evaluate double integral over planar region
quadgk Numerically evaluate integral, adaptive Gauss-Kronrod quadrature
quadl Numerically evaluate integral, adaptive Lobatto quadrature
quadv Vectorized quadrature
triplequad Numerically evaluate triple integral

**Specialized Math**

airy Airy functions
besselh Bessel function of third kind (Hankel function)
besseli Modified Bessel function of first kind
besselj Bessel function of first kind
besselk Modified Bessel function of second kind
bessely Bessel function of second kind
<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td>Beta function</td>
</tr>
<tr>
<td>betainc</td>
<td>Incomplete beta function</td>
</tr>
<tr>
<td>betaincinv</td>
<td>Beta inverse cumulative distribution function</td>
</tr>
<tr>
<td>betaln</td>
<td>Logarithm of beta function</td>
</tr>
<tr>
<td>ellipj</td>
<td>Jacobi elliptic functions</td>
</tr>
<tr>
<td>ellipke</td>
<td>Complete elliptic integrals of first and second kind</td>
</tr>
<tr>
<td>erf, erfc, erfcx, erfinv, erfcinv</td>
<td>Error functions</td>
</tr>
<tr>
<td>expint</td>
<td>Exponential integral</td>
</tr>
<tr>
<td>gamma, gammainc, gammaln</td>
<td>Gamma functions</td>
</tr>
<tr>
<td>gammaincinv</td>
<td>Inverse incomplete gamma function</td>
</tr>
<tr>
<td>legendre</td>
<td>Associated Legendre functions</td>
</tr>
<tr>
<td>psi</td>
<td>Psi (polygamma) function</td>
</tr>
</tbody>
</table>

**Sparse Matrices**

- **Elementary Sparse Matrices** (p. 1-38)
  - Create random and nonrandom sparse matrices
- **Full to Sparse Conversion** (p. 1-38)
  - Convert full matrix to sparse, sparse matrix to full
- **Sparse Matrix Manipulation** (p. 1-39)
  - Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern
- **Reordering Algorithms** (p. 1-39)
  - Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations
## Function Reference

### Linear Algebra (p. 1-40)
- Compute norms, eigenvalues, factorizations, least squares, structural rank

### Linear Equations (Iterative Methods) (p. 1-40)
- Methods for conjugate and biconjugate gradients, residuals, lower quartile

### Tree Operations (p. 1-41)
- Elimination trees, tree plotting, factorization analysis

### Elementary Sparse Matrices
- **spdiags**
  - Extract and create sparse band and diagonal matrices
- **speye**
  - Sparse identity matrix
- **sprand**
  - Sparse uniformly distributed random matrix
- **sprandn**
  - Sparse normally distributed random matrix
- **sprandsym**
  - Sparse symmetric random matrix

### Full to Sparse Conversion
- **find**
  - Find indices and values of nonzero elements
- **full**
  - Convert sparse matrix to full matrix
- **sparse**
  - Create sparse matrix
- **spconvert**
  - Import matrix from sparse matrix external format
Sparse Matrix Manipulation

- **issparse**: Determine whether input is sparse
- **nnz**: Number of nonzero matrix elements
- **nonzeros**: Nonzero matrix elements
- **nzmax**: Amount of storage allocated for nonzero matrix elements
- **spalloc**: Allocate space for sparse matrix
- **spfun**: Apply function to nonzero sparse matrix elements
- **spones**: Replace nonzero sparse matrix elements with ones
- **spparms**: Set parameters for sparse matrix routines
- **spy**: Visualize sparsity pattern

Reordering Algorithms

- **amd**: Approximate minimum degree permutation
- **colamd**: Column approximate minimum degree permutation
- **colperm**: Sparse column permutation based on nonzero count
- **dmperm**: Dulmage-Mendelsohn decomposition
- **ldl**: Block LDL’ factorization for Hermitian indefinite matrices
- **randperm**: Random permutation
- **symamd**: Symmetric approximate minimum degree permutation
- **symrcm**: Sparse reverse Cuthill-McKee ordering
**Linear Algebra**

- **cholinc**
  - Sparse incomplete Cholesky and Cholesky-Infinity factorizations
- **condest**
  - 1-norm condition number estimate
- **eigs**
  - Largest eigenvalues and eigenvectors of matrix
- **ilu**
  - Sparse incomplete LU factorization
- **luinc**
  - Sparse incomplete LU factorization
- **normest**
  - 2-norm estimate
- **spaugment**
  - Form least squares augmented system
- **sprank**
  - Structural rank
- **svds**
  - Find singular values and vectors

**Linear Equations (Iterative Methods)**

- **bicg**
  - Biconjugate gradients method
- **bicgstab**
  - Biconjugate gradients stabilized method
- **bicgstabl**
  - Biconjugate gradients stabilized (l) method
- **cgs**
  - Conjugate gradients squared method
- **gmres**
  - Generalized minimum residual method (with restarts)
- **lsqr**
  - LSQR method
- **minres**
  - Minimum residual method
- **pcg**
  - Preconditioned conjugate gradients method
- **qmr**
  - Quasi-minimal residual method
symmlq  Symmetric LQ method
tfqmr  Transpose-free quasi-minimal residual method

**Tree Operations**

etree  Elimination tree
etreeplot  Plot elimination tree
gplot  Plot nodes and links representing adjacency matrix
symbfact  Symbolic factorization analysis
treelayout  Lay out tree or forest
treeplot  Plot picture of tree
unmesh  Convert edge matrix to coordinate and Laplacian matrices

**Math Constants**

eps  Floating-point relative accuracy
i  Imaginary unit
Inf  Infinity
intmax  Largest value of specified integer type
intmin  Smallest value of specified integer type
j  Imaginary unit
NaN  Not-a-Number
pi  Ratio of circle’s circumference to its diameter, π
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>realmax</code></td>
<td>Largest positive floating-point number</td>
</tr>
<tr>
<td><code>realmin</code></td>
<td>Smallest positive normalized floating-point number</td>
</tr>
</tbody>
</table>
Data Analysis

Basic Operations (p. 1-43)
Descriptive Statistics (p. 1-43)
Filtering and Convolution (p. 1-44)
Interpolation and Regression (p. 1-44)
Fourier Transforms (p. 1-45)
Derivatives and Integrals (p. 1-45)
Time Series Objects (p. 1-46)
Time Series Collections (p. 1-49)

Basic Operations

- brush: Interactively mark, delete, modify, and save observations in graphs
- cumprod: Cumulative product
- cumsum: Cumulative sum
- linkdata: Automatically update graphs when variables change
- prod: Product of array elements
- sort: Sort array elements in ascending or descending order
- sortrows: Sort rows in ascending order
- sum: Sum of array elements

Descriptive Statistics

- corrcoef: Correlation coefficients
- cov: Covariance matrix
### Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>max</strong></td>
<td>Largest elements in array</td>
</tr>
<tr>
<td><strong>mean</strong></td>
<td>Average or mean value of array</td>
</tr>
<tr>
<td><strong>median</strong></td>
<td>Median value of array</td>
</tr>
<tr>
<td><strong>min</strong></td>
<td>Smallest elements in array</td>
</tr>
<tr>
<td><strong>mode</strong></td>
<td>Most frequent values in array</td>
</tr>
<tr>
<td><strong>std</strong></td>
<td>Standard deviation</td>
</tr>
<tr>
<td><strong>var</strong></td>
<td>Variance</td>
</tr>
</tbody>
</table>

### Filtering and Convolution

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>conv</strong></td>
<td>Convolution and polynomial multiplication</td>
</tr>
<tr>
<td><strong>conv2</strong></td>
<td>2-D convolution</td>
</tr>
<tr>
<td><strong>convn</strong></td>
<td>N-D convolution</td>
</tr>
<tr>
<td><strong>deconv</strong></td>
<td>Deconvolution and polynomial division</td>
</tr>
<tr>
<td><strong>detrend</strong></td>
<td>Remove linear trends</td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td>1-D digital filter</td>
</tr>
<tr>
<td><strong>filter2</strong></td>
<td>2-D digital filter</td>
</tr>
</tbody>
</table>

### Interpolation and Regression

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>interp1</strong></td>
<td>1-D data interpolation (table lookup)</td>
</tr>
<tr>
<td><strong>interp2</strong></td>
<td>2-D data interpolation (table lookup)</td>
</tr>
<tr>
<td><strong>interp3</strong></td>
<td>3-D data interpolation (table lookup)</td>
</tr>
<tr>
<td><strong>interpn</strong></td>
<td>N-D data interpolation (table lookup)</td>
</tr>
</tbody>
</table>
| **mldivide \
| **mrdivide /** | Left or right matrix division |
| **polyfit** | Polynomial curve fitting |
| **polyval** | Polynomial evaluation |
### Fourier Transforms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Absolute value and complex magnitude</td>
</tr>
<tr>
<td>angle</td>
<td>Phase angle</td>
</tr>
<tr>
<td>cplxpair</td>
<td>Sort complex numbers into complex conjugate pairs</td>
</tr>
<tr>
<td>fft</td>
<td>Discrete Fourier transform</td>
</tr>
<tr>
<td>fft2</td>
<td>2-D discrete Fourier transform</td>
</tr>
<tr>
<td>fftn</td>
<td>N-D discrete Fourier transform</td>
</tr>
<tr>
<td>fftshift</td>
<td>Shift zero-frequency component to center of spectrum</td>
</tr>
<tr>
<td>fftw</td>
<td>Interface to FFTW library run-time algorithm tuning control</td>
</tr>
<tr>
<td>ifft</td>
<td>Inverse discrete Fourier transform</td>
</tr>
<tr>
<td>ifft2</td>
<td>2-D inverse discrete Fourier transform</td>
</tr>
<tr>
<td>ifftn</td>
<td>N-D inverse discrete Fourier transform</td>
</tr>
<tr>
<td>ifftshift</td>
<td>Inverse FFT shift</td>
</tr>
<tr>
<td>nextpow2</td>
<td>Next higher power of 2</td>
</tr>
<tr>
<td>unwrap</td>
<td>Correct phase angles to produce smoother phase plots</td>
</tr>
</tbody>
</table>

### Derivatives and Integrals

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cumtrapz</td>
<td>Cumulative trapezoidal numerical integration</td>
</tr>
<tr>
<td>del2</td>
<td>Discrete Laplacian</td>
</tr>
<tr>
<td>diff</td>
<td>Differences and approximate derivatives</td>
</tr>
</tbody>
</table>
gradient
polyder
polyint
trapz

**Time Series Objects**

Utilities (p. 1-46)
Combine *timeseries* objects, query and set *timeseries* object properties, plot *timeseries* objects

Data Manipulation (p. 1-47)
Add or delete data, manipulate *timeseries* objects

Event Data (p. 1-48)
Add or delete events, create new *timeseries* objects based on event data

Descriptive Statistics (p. 1-48)
Descriptive statistics for *timeseries* objects

**Utilities**

get (timeseries)
Query *timeseries* object property values

getdatasamplesize
Size of data sample in *timeseries* object

getqualitydesc
Data quality descriptions

isempty (timeseries)
Determine whether *timeseries* object is empty

length (timeseries)
Length of time vector

plot (timeseries)
Plot time series

set (timeseries)
Set properties of *timeseries* object

size (timeseries)
Size of *timeseries* object
### Data Analysis

- **timeseries**: Create timeseries object
- **tsdata.event**: Construct event object for timeseries object
- **tsprops**: Help on timeseries object properties
- **tstool**: Open Time Series Tools GUI

### Data Manipulation

- **addsample**: Add data sample to timeseries object
- **ctranspose (timeseries)**: Transpose timeseries object
- **delsample**: Remove sample from timeseries object
- **detrend (timeseries)**: Subtract mean or best-fit line and all NaNs from time series
- **filter (timeseries)**: Shape frequency content of time series
- **getabstime (timeseries)**: Extract date-string time vector into cell array
- **getinterpmethod**: Interpolation method for timeseries object
- **getsampleusingtime (timeseries)**: Extract data samples into new timeseries object
- **idealfilter (timeseries)**: Apply ideal (noncausal) filter to timeseries object
- **resample (timeseries)**: Select or interpolate timeseries data using new time vector
- **setabstime (timeseries)**: Set times of timeseries object as date strings
- **setinterpmethod**: Set default interpolation method for timeseries object
### Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronize</td>
<td>Synchronize and resample two timeseries objects using common time vector</td>
</tr>
<tr>
<td>transpose (timeseries)</td>
<td>Transpose timeseries object</td>
</tr>
<tr>
<td>vertcat (timeseries)</td>
<td>Vertical concatenation of timeseries objects</td>
</tr>
</tbody>
</table>

#### Event Data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addevent</td>
<td>Add event to timeseries object</td>
</tr>
<tr>
<td>delevent</td>
<td>Remove tsdata.event objects from timeseries object</td>
</tr>
<tr>
<td>getsafterevent</td>
<td>New timeseries object with samples occurring at or after event</td>
</tr>
<tr>
<td>getsafteratevent</td>
<td>New timeseries object with samples occurring after event</td>
</tr>
<tr>
<td>getsatevent</td>
<td>New timeseries object with samples occurring at event</td>
</tr>
<tr>
<td>gettsbeforeatevent</td>
<td>New timeseries object with samples occurring before or at event</td>
</tr>
<tr>
<td>gettsbeforeevent</td>
<td>New timeseries object with samples occurring before event</td>
</tr>
<tr>
<td>gettsbetweenevents</td>
<td>New timeseries object with samples occurring between events</td>
</tr>
</tbody>
</table>

#### Descriptive Statistics

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iqr (timeseries)</td>
<td>Interquartile range of timeseries data</td>
</tr>
<tr>
<td>max (timeseries)</td>
<td>Maximum value of timeseries data</td>
</tr>
<tr>
<td>mean (timeseries)</td>
<td>Mean value of timeseries data</td>
</tr>
<tr>
<td>median (timeseries)</td>
<td>Median value of timeseries data</td>
</tr>
</tbody>
</table>
min (timeseries)  Minimum value of timeseries data
std (timeseries) Standard deviation of timeseries data
sum (timeseries) Sum of timeseries data
var (timeseries) Variance of timeseries data

Time Series Collections

Utilities (p. 1-49) Query and set tscollection object properties, plot tscollection objects
Data Manipulation (p. 1-50) Add or delete data, manipulate tscollection objects

Utilities

get (tscollection) Query tscollection object property values
isempty (tscollection) Determine whether tscollection object is empty
length (tscollection) Length of time vector
plot (timeseries) Plot time series
set (tscollection) Set properties of tscollection object
size (tscollection) Size of tscollection object
tscollection Create tscollection object
tstool Open Time Series Tools GUI
## Data Manipulation

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addsampletocollection</td>
<td>Add sample to <code>tscollection</code> object</td>
</tr>
<tr>
<td>addts</td>
<td>Add <code>timeseries</code> object to <code>tscollection</code> object</td>
</tr>
<tr>
<td>delsamplefromcollection</td>
<td>Remove sample from <code>tscollection</code> object</td>
</tr>
<tr>
<td>getabstime(tscollection)</td>
<td>Extract date-string time vector into cell array</td>
</tr>
<tr>
<td>getsampleusingtime(tscollection)</td>
<td>Extract data samples into new <code>tscollection</code> object</td>
</tr>
<tr>
<td>gettimeseriesnames</td>
<td>Cell array of names of <code>timeseries</code> objects in <code>tscollection</code> object</td>
</tr>
<tr>
<td>horzcat(tscollection)</td>
<td>Horizontal concatenation for <code>tscollection</code> objects</td>
</tr>
<tr>
<td>removets</td>
<td>Remove <code>timeseries</code> objects from <code>tscollection</code> object</td>
</tr>
<tr>
<td>resample(tscollection)</td>
<td>Select or interpolate data in <code>tscollection</code> using new time vector</td>
</tr>
<tr>
<td>setabstime(tscollection)</td>
<td>Set times of <code>tscollection</code> object as date strings</td>
</tr>
<tr>
<td>settimeseriesnames</td>
<td>Change name of <code>timeseries</code> object in <code>tscollection</code></td>
</tr>
<tr>
<td>vertcat(tscollection)</td>
<td>Vertical concatenation for <code>tscollection</code> objects</td>
</tr>
</tbody>
</table>
Programming and Data Types

Data Types (p. 1-51)
Numeric, character, structures, cell arrays, and data type conversion

Data Type Conversion (p. 1-59)
Convert one numeric type to another, numeric to string, string to numeric, structure to cell array, etc.

Operators and Special Characters (p. 1-61)
Arithmetic, relational, and logical operators, and special characters

Strings (p. 1-64)
Create, identify, manipulate, parse, evaluate, and compare strings

Bit-Wise Operations (p. 1-67)
Perform set, shift, and, or, compare, etc. on specific bit fields

Logical Operations (p. 1-67)
Evaluate conditions, testing for true or false

Relational Operations (p. 1-68)
Compare values for equality, greater than, less than, etc.

Set Operations (p. 1-68)
Find set members, unions, intersections, etc.

Date and Time Operations (p. 1-69)
Obtain information about dates and times

Programming in MATLAB (p. 1-69)
M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling

Data Types

Numeric Types (p. 1-52)
Integer and floating-point data

Characters and Strings (p. 1-53)
Characters and arrays of characters

Structures (p. 1-54)
Data of varying types and sizes stored in fields of a structure
<table>
<thead>
<tr>
<th>Function Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Arrays (p. 1-55)</td>
</tr>
<tr>
<td>Function Handles (p. 1-56)</td>
</tr>
<tr>
<td>Java Classes and Objects (p. 1-56)</td>
</tr>
<tr>
<td>Data Type Identification (p. 1-58)</td>
</tr>
</tbody>
</table>

**Numeric Types**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrayfun</td>
<td>Apply function to each element of array</td>
</tr>
<tr>
<td>cast</td>
<td>Cast variable to different data type</td>
</tr>
<tr>
<td>cat</td>
<td>Concatenate arrays along specified dimension</td>
</tr>
<tr>
<td>class</td>
<td>Create object or return class of object</td>
</tr>
<tr>
<td>find</td>
<td>Find indices and values of nonzero elements</td>
</tr>
<tr>
<td>intmax</td>
<td>Largest value of specified integer type</td>
</tr>
<tr>
<td>intmin</td>
<td>Smallest value of specified integer type</td>
</tr>
<tr>
<td>intwarning</td>
<td>Control state of integer warnings</td>
</tr>
<tr>
<td>ipermute</td>
<td>Inverse permute dimensions of N-D array</td>
</tr>
<tr>
<td>isa</td>
<td>Determine whether input is object of given class</td>
</tr>
<tr>
<td>isequal</td>
<td>Test arrays for equality</td>
</tr>
<tr>
<td>isequalwithequalnans</td>
<td>Test arrays for equality, treating NaNs as equal</td>
</tr>
<tr>
<td>isfinite</td>
<td>Array elements that are finite</td>
</tr>
</tbody>
</table>
### Programming and Data Types

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isinf</td>
<td>Array elements that are infinite</td>
</tr>
<tr>
<td>isnan</td>
<td>Array elements that are NaN</td>
</tr>
<tr>
<td>isnumeric</td>
<td>Determine whether input is numeric array</td>
</tr>
<tr>
<td>isreal</td>
<td>Check if input is real array</td>
</tr>
<tr>
<td>isscalar</td>
<td>Determine whether input is scalar</td>
</tr>
<tr>
<td>isvector</td>
<td>Determine whether input is vector</td>
</tr>
<tr>
<td>permute</td>
<td>Rearrange dimensions of N-D array</td>
</tr>
<tr>
<td>realmax</td>
<td>Largest positive floating-point number</td>
</tr>
<tr>
<td>realmin</td>
<td>Smallest positive normalized floating-point number</td>
</tr>
<tr>
<td>reshape</td>
<td>Reshape array</td>
</tr>
<tr>
<td>squeeze</td>
<td>Remove singleton dimensions</td>
</tr>
<tr>
<td>zeros</td>
<td>Create array of all zeros</td>
</tr>
</tbody>
</table>

### Characters and Strings

See “Strings” on page 1-64 for all string-related functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellstr</td>
<td>Create cell array of strings from character array</td>
</tr>
<tr>
<td>char</td>
<td>Convert to character array (string)</td>
</tr>
<tr>
<td>eval</td>
<td>Execute string containing MATLAB expression</td>
</tr>
<tr>
<td>findstr</td>
<td>Find string within another, longer string</td>
</tr>
<tr>
<td>isstr</td>
<td>Determine whether input is character array</td>
</tr>
<tr>
<td>regexp, regexpi</td>
<td>Match regular expression</td>
</tr>
<tr>
<td>sprintf</td>
<td>Write formatted data to string</td>
</tr>
</tbody>
</table>
sscanf
Read formatted data from string

strcat
Concatenate strings horizontally

strcmp, strcmpi
Compare strings

strings
String handling

strjust
Justify character array

strmatch
Find possible matches for string

strread
Read formatted data from string

strrep
Find and replace substring

strtrim
Remove leading and trailing white space from string

strvcat
Concatenate strings vertically

**Structures**

arrayfun
Apply function to each element of array

cell2struct
Convert cell array to structure array

class
Create object or return class of object

deal
Distribute inputs to outputs

fieldnames
Field names of structure, or public fields of object

getfield
Field of structure array

isa
Determine whether input is object of given class

isequal
Test arrays for equality

isfield
Determine whether input is structure array field

isscalar
Determine whether input is scalar

isstruct
Determine whether input is structure array
### Programming and Data Types

- **isvector** Determine whether input is vector
- **orderfields** Order fields of structure array
- **rmfield** Remove fields from structure
- **setfield** Set value of structure array field
- **struct** Create structure array
- **struct2cell** Convert structure to cell array
- **structfun** Apply function to each field of scalar structure

#### Cell Arrays

- **cell** Construct cell array
- **cell2mat** Convert cell array of matrices to single matrix
- **cell2struct** Convert cell array to structure array
- **celldisp** Cell array contents
- **cellfun** Apply function to each cell in cell array
- **cellplot** Graphically display structure of cell array
- **cellstr** Create cell array of strings from character array
- **class** Create object or return class of object
- **deal** Distribute inputs to outputs
- **isa** Determine whether input is object of given class
- **iscell** Determine whether input is cell array
- **iscellstr** Determine whether input is cell array of strings
isequal
isscalar
isvector
mat2cell
num2cell
struct2cell

Test arrays for equality
Determine whether input is scalar
Determine whether input is vector
Divide matrix into cell array of matrices
Convert numeric array to cell array
Convert structure to cell array

Function Handles

class
feval
func2str
functions
function_handle (@)
isa
isequal
str2func

Create object or return class of object
Evaluate function
Construct function name string from function handle
Information about function handle
Handle used in calling functions indirectly
Determine whether input is object of given class
Test arrays for equality
Construct function handle from function name string

Java Classes and Objects

cell
class
clear
depfun

Construct cell array
Create object or return class of object
Remove items from workspace, freeing up system memory
List dependencies of M-file or P-file
<table>
<thead>
<tr>
<th>exist</th>
<th>Check existence of variable, function, directory, or Java programming language class</th>
</tr>
</thead>
<tbody>
<tr>
<td>fieldnames</td>
<td>Field names of structure, or public fields of object</td>
</tr>
<tr>
<td>im2java</td>
<td>Convert image to Java image</td>
</tr>
<tr>
<td>import</td>
<td>Add package or class to current import list</td>
</tr>
<tr>
<td>inmem</td>
<td>Names of M-files, MEX-files, Sun Java classes in memory</td>
</tr>
<tr>
<td>isa</td>
<td>Determine whether input is object of given class</td>
</tr>
<tr>
<td>isjava</td>
<td>Determine whether input is Sun Java object</td>
</tr>
<tr>
<td>javaaddpath</td>
<td>Add entries to dynamic Sun Java class path</td>
</tr>
<tr>
<td>javaArray</td>
<td>Construct Sun Java array</td>
</tr>
<tr>
<td>javachk</td>
<td>Generate error message based on Sun Java feature support</td>
</tr>
<tr>
<td>javaclasspath</td>
<td>Get and set Sun Java class path</td>
</tr>
<tr>
<td>javaMethod</td>
<td>Call Sun Java method</td>
</tr>
<tr>
<td>javaMethodEDT</td>
<td>Call Sun Java method from Event Dispatch Thread (EDT)</td>
</tr>
<tr>
<td>javaObject</td>
<td>Construct Sun Java object</td>
</tr>
<tr>
<td>javaObjectEDT</td>
<td>Construct Sun Java object on Event Dispatch Thread (EDT)</td>
</tr>
<tr>
<td>javarmpath</td>
<td>Remove entries from dynamic Sun Java class path</td>
</tr>
<tr>
<td>methods</td>
<td>Information on class methods</td>
</tr>
<tr>
<td>methodsvindow</td>
<td>Information on class methods in separate window</td>
</tr>
</tbody>
</table>
### usejava
Determine whether Sun Java feature is supported in MATLAB software

### which
Locate functions and files

### Data Type Identification

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is*</td>
<td>Detect state</td>
</tr>
<tr>
<td>isa</td>
<td>Determine whether input is object of given class</td>
</tr>
<tr>
<td>iscell</td>
<td>Determine whether input is cell array</td>
</tr>
<tr>
<td>iscellstr</td>
<td>Determine whether input is cell array of strings</td>
</tr>
<tr>
<td>ischar</td>
<td>Determine whether item is character array</td>
</tr>
<tr>
<td>isfield</td>
<td>Determine whether input is structure array field</td>
</tr>
<tr>
<td>isfloat</td>
<td>Determine whether input is floating-point array</td>
</tr>
<tr>
<td>isinteger</td>
<td>Determine whether input is integer array</td>
</tr>
<tr>
<td>isjava</td>
<td>Determine whether input is Sun Java object</td>
</tr>
<tr>
<td>islogical</td>
<td>Determine whether input is logical array</td>
</tr>
<tr>
<td>isnumeric</td>
<td>Determine whether input is numeric array</td>
</tr>
<tr>
<td>isobject</td>
<td>Determine if input is MATLAB object</td>
</tr>
<tr>
<td>isreal</td>
<td>Check if input is real array</td>
</tr>
<tr>
<td>isstr</td>
<td>Determine whether input is character array</td>
</tr>
</tbody>
</table>
isstruct
validateattributes
who, whos

**Data Type Conversion**

Numeric (p. 1-59)
String to Numeric (p. 1-60)
Numeric to String (p. 1-60)
Other Conversions (p. 1-61)

**Numeric**

cast
double
int8, int16, int32, int64
single
typecast
uint8, uint16, uint32, uint64

Determine whether input is structure array
Check validity of array
List variables in workspace

Convert data of one numeric type to another numeric type
Convert characters to numeric equivalent
Convert numeric to character equivalent
Convert to structure, cell array, function handle, etc.

Cast variable to different data type
Convert to double precision
Convert to signed integer
Convert to single precision
Convert data types without changing underlying data
Convert to unsigned integer
### String to Numeric

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base2dec</td>
<td>Convert base N number string to decimal number</td>
</tr>
<tr>
<td>bin2dec</td>
<td>Convert binary number string to decimal number</td>
</tr>
<tr>
<td>cast</td>
<td>Cast variable to different data type</td>
</tr>
<tr>
<td>hex2dec</td>
<td>Convert hexadecimal number string to decimal number</td>
</tr>
<tr>
<td>hex2num</td>
<td>Convert hexadecimal number string to double-precision number</td>
</tr>
<tr>
<td>str2double</td>
<td>Convert string to double-precision value</td>
</tr>
<tr>
<td>str2num</td>
<td>Convert string to number</td>
</tr>
</tbody>
</table>
| unicode2native | Convert Unicode® characters to numeric bytes |}

### Numeric to String

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cast</td>
<td>Cast variable to different data type</td>
</tr>
<tr>
<td>char</td>
<td>Convert to character array (string)</td>
</tr>
<tr>
<td>dec2base</td>
<td>Convert decimal to base N number in string</td>
</tr>
<tr>
<td>dec2bin</td>
<td>Convert decimal to binary number in string</td>
</tr>
<tr>
<td>dec2hex</td>
<td>Convert decimal to hexadecimal number in string</td>
</tr>
<tr>
<td>int2str</td>
<td>Convert integer to string</td>
</tr>
<tr>
<td>mat2str</td>
<td>Convert matrix to string</td>
</tr>
<tr>
<td>native2unicode</td>
<td>Convert numeric bytes to Unicode characters</td>
</tr>
<tr>
<td>num2str</td>
<td>Convert number to string</td>
</tr>
</tbody>
</table>
**Other Conversions**

- **cell2mat**: Convert cell array of matrices to single matrix
- **cell2struct**: Convert cell array to structure array
- **datestr**: Convert date and time to string format
- **func2str**: Construct function name string from function handle
- **logical**: Convert numeric values to logical
- **mat2cell**: Divide matrix into cell array of matrices
- **num2cell**: Convert numeric array to cell array
- **num2hex**: Convert singles and doubles to IEEE® hexadecimal strings
- **str2func**: Construct function handle from function name string
- **str2mat**: Form blank-padded character matrix from strings
- **struct2cell**: Convert structure to cell array

**Operators and Special Characters**

- **Arithmetic Operators** (p. 1-62): Plus, minus, power, left and right divide, transpose, etc.
- **Relational Operators** (p. 1-62): Equal to, greater than, less than or equal to, etc.
- **Logical Operators** (p. 1-62): Element-wise and short circuit and, or, not
- **Special Characters** (p. 1-63): Array constructors, line continuation, comments, etc.
**Arithmetic Operators**

+    Plus
-    Minus
.    Decimal point
=    Assignment
*    Matrix multiplication
/    Matrix right division
\    Matrix left division
^    Matrix power
'    Matrix transpose
.*   Array multiplication (element-wise)
./   Array right division (element-wise)
.\   Array left division (element-wise)
.^   Array power (element-wise)
.'   Array transpose

**Relational Operators**

<    Less than
<=   Less than or equal to
>
>=   Greater than or equal to
==   Equal to
~=   Not equal to

**Logical Operators**

See also “Logical Operations” on page 1-67 for functions like xor, all, any, etc.
&& Logical AND
|| Logical OR
& Logical AND for arrays
| Logical OR for arrays
~ Logical NOT

**Special Characters**

: Create vectors, subscript arrays, specify for-loop iterations
( ) Pass function arguments, prioritize operators
[ ] Construct array, concatenate elements, specify multiple outputs from function
{} Construct cell array, index into cell array
. Insert decimal point, define structure field, reference methods of object
.( ) Reference dynamic field of structure
.. Reference parent directory
...
... Continue statement to next line
, Separate rows of array, separate function input/output arguments, separate commands
; Separate columns of array, suppress output from current command
% Insert comment line into code
%
%{ %} Insert block of comments into code
! Issue command to operating system
'' Construct character array
@ Construct function handle, reference class directory
# Strings

**Description of Strings in MATLAB** (p. 1-64)

- Basics of string handling in MATLAB
- String Creation (p. 1-64)
  - Create strings, cell arrays of strings, concatenate strings together
- String Identification (p. 1-65)
  - Identify characteristics of strings
- String Manipulation (p. 1-65)
  - Convert case, strip blanks, replace characters
- String Parsing (p. 1-66)
  - Formatted read, regular expressions, locate substrings
- String Evaluation (p. 1-66)
  - Evaluate stated expression in string
- String Comparison (p. 1-66)
  - Compare contents of strings

## Description of Strings in MATLAB

- `strings`  
  - String handling

## String Creation

- `blanks`  
  - Create string of blank characters
- `cellstr`  
  - Create cell array of strings from character array
- `char`  
  - Convert to character array (string)
- `sprintf`  
  - Write formatted data to string
- `strcat`  
  - Concatenate strings horizontally
- `strvcat`  
  - Concatenate strings vertically
### String Identification

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Create object or return class of object</td>
</tr>
<tr>
<td>isa</td>
<td>Determine whether input is object of given class</td>
</tr>
<tr>
<td>iscellstr</td>
<td>Determine whether input is cell array of strings</td>
</tr>
<tr>
<td>ischar</td>
<td>Determine whether item is character array</td>
</tr>
<tr>
<td>isletter</td>
<td>Array elements that are alphabetic letters</td>
</tr>
<tr>
<td>isscalar</td>
<td>Determine whether input is scalar</td>
</tr>
<tr>
<td>isspace</td>
<td>Array elements that are space characters</td>
</tr>
<tr>
<td>isstrprop</td>
<td>Determine whether string is of specified category</td>
</tr>
<tr>
<td>isvector</td>
<td>Determine whether input is vector</td>
</tr>
<tr>
<td>validatestring</td>
<td>Check validity of text string</td>
</tr>
</tbody>
</table>

### String Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deblank</td>
<td>Strip trailing blanks from end of string</td>
</tr>
<tr>
<td>lower</td>
<td>Convert string to lowercase</td>
</tr>
<tr>
<td>strjust</td>
<td>Justify character array</td>
</tr>
<tr>
<td>strrep</td>
<td>Find and replace substring</td>
</tr>
<tr>
<td>strtrim</td>
<td>Remove leading and trailing white space from string</td>
</tr>
<tr>
<td>upper</td>
<td>Convert string to uppercase</td>
</tr>
</tbody>
</table>
### String Parsing

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findstr</td>
<td>Find string within another, longer string</td>
</tr>
<tr>
<td>regexp, regexpi</td>
<td>Match regular expression</td>
</tr>
<tr>
<td>regexprep</td>
<td>Replace string using regular expression</td>
</tr>
<tr>
<td>regexptranslate</td>
<td>Translate string into regular expression</td>
</tr>
<tr>
<td>sscanf</td>
<td>Read formatted data from string</td>
</tr>
<tr>
<td>strfind</td>
<td>Find one string within another</td>
</tr>
<tr>
<td>strread</td>
<td>Read formatted data from string</td>
</tr>
<tr>
<td>strtok</td>
<td>Selected parts of string</td>
</tr>
</tbody>
</table>

### String Evaluation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval</td>
<td>Execute string containing MATLAB expression</td>
</tr>
<tr>
<td>evalc</td>
<td>Evaluate MATLAB expression with capture</td>
</tr>
<tr>
<td>evalin</td>
<td>Execute MATLAB expression in specified workspace</td>
</tr>
</tbody>
</table>

### String Comparison

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcmp, strcmpi</td>
<td>Compare strings</td>
</tr>
<tr>
<td>strmatch</td>
<td>Find possible matches for string</td>
</tr>
<tr>
<td>strncmp, strncmp</td>
<td>Compare first n characters of strings</td>
</tr>
</tbody>
</table>
Bit-Wise Operations

- `bitand` Bitwise AND
- `bitcmp` Bitwise complement
- `bitget` Bit at specified position
- `bitmax` Maximum double-precision floating-point integer
- `bitor` Bitwise OR
- `bitset` Set bit at specified position
- `bitshift` Shift bits specified number of places
- `bitxor` Bitwise XOR
- `swapbytes` Swap byte ordering

Logical Operations

- `all` Determine whether all array elements are nonzero
- `and` Find logical AND of array or scalar inputs
- `any` Determine whether any array elements are nonzero
- `false` Logical 0 (false)
- `find` Find indices and values of nonzero elements
- `isa` Determine whether input is object of given class
- `iskeyword` Determine whether input is MATLAB keyword
- `isvarname` Determine whether input is valid variable name
- `logical` Convert numeric values to logical
not Find logical NOT of array or scalar input
or Find logical OR of array or scalar inputs
true Logical 1 (true)
xor Logical exclusive-OR

See “Operators and Special Characters” on page 1-61 for logical operators.

**Relational Operations**

eq Test for equality
ge Test for greater than or equal to
gt Test for greater than
le Test for less than or equal to
lt Test for less than
ne Test for inequality

See “Operators and Special Characters” on page 1-61 for relational operators.

**Set Operations**

intersect Find set intersection of two vectors
ismember Array elements that are members of set
issorted Determine whether set elements are in sorted order
setdiff Find set difference of two vectors
setxor Find set exclusive OR of two vectors
union
unique

Date and Time Operations

addtodate
calendar
clock
cputime
date
datenum
datestr
datevec
eomday
etime
now
weekday

Programming in MATLAB

M-Files and Scripts (p. 1-70)
Evaluation (p. 1-71)
Timer (p. 1-72)
### Variables and Functions in Memory (p. 1-73)
- List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches

### Control Flow (p. 1-74)
- if-then-else, for loops, switch-case, try-catch

### Error Handling (p. 1-75)
- Generate warnings and errors, test for and catch errors, retrieve most recent error message

### MEX Programming (p. 1-76)
- Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

### M-Files and Scripts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addOptional (inputParser)</td>
<td>Add optional argument to inputParser schema</td>
</tr>
<tr>
<td>addParamValue (inputParser)</td>
<td>Add parameter-value argument to inputParser schema</td>
</tr>
<tr>
<td>addRequired (inputParser)</td>
<td>Add required argument to inputParser schema</td>
</tr>
<tr>
<td>createCopy (inputParser)</td>
<td>Create copy of inputParser object</td>
</tr>
<tr>
<td>depdir</td>
<td>List dependent directories of M-file or P-file</td>
</tr>
<tr>
<td>depfun</td>
<td>List dependencies of M-file or P-file</td>
</tr>
<tr>
<td>echo</td>
<td>Echo M-files during execution</td>
</tr>
<tr>
<td>end</td>
<td>Terminate block of code, or indicate last array index</td>
</tr>
<tr>
<td>function</td>
<td>Declare M-file function</td>
</tr>
<tr>
<td>input</td>
<td>Request user input</td>
</tr>
<tr>
<td>inputname</td>
<td>Variable name of function input</td>
</tr>
<tr>
<td>inputParser</td>
<td>Construct input parser object</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>mfilename</td>
<td>Name of currently running M-file</td>
</tr>
<tr>
<td>namelengthmax</td>
<td>Maximum identifier length</td>
</tr>
<tr>
<td>nargchk</td>
<td>Validate number of input arguments</td>
</tr>
<tr>
<td>nargin, nargout</td>
<td>Number of function arguments</td>
</tr>
<tr>
<td>nargoutchk</td>
<td>Validate number of output arguments</td>
</tr>
<tr>
<td>parse (inputParser)</td>
<td>Parse and validate named inputs</td>
</tr>
<tr>
<td>pcode</td>
<td>Create protected M-file (P-file)</td>
</tr>
<tr>
<td>script</td>
<td>Script M-file description</td>
</tr>
<tr>
<td>syntax</td>
<td>Two ways to call MATLAB functions</td>
</tr>
<tr>
<td>varargin</td>
<td>Variable length input argument list</td>
</tr>
<tr>
<td>varargout</td>
<td>Variable length output argument list</td>
</tr>
</tbody>
</table>

**Evaluation**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ans</td>
<td>Most recent answer</td>
</tr>
<tr>
<td>arrayfun</td>
<td>Apply function to each element of array</td>
</tr>
<tr>
<td>assert</td>
<td>Generate error when condition is violated</td>
</tr>
<tr>
<td>builtin</td>
<td>Execute built-in function from overloaded method</td>
</tr>
<tr>
<td>cellfun</td>
<td>Apply function to each cell in cell array</td>
</tr>
<tr>
<td>echo</td>
<td>Echo M-files during execution</td>
</tr>
<tr>
<td>eval</td>
<td>Execute string containing MATLAB expression</td>
</tr>
<tr>
<td>evalc</td>
<td>Evaluate MATLAB expression with capture</td>
</tr>
</tbody>
</table>
evalin  
Execute MATLAB expression in specified workspace

feval  
Evaluate function

iskeyword  
Determine whether input is MATLAB keyword

isvarname  
Determine whether input is valid variable name

pause  
Halt execution temporarily

run  
Run script that is not on current path

script  
Script M-file description

structfun  
Apply function to each field of scalar structure

symvar  
Determine symbolic variables in expression

tic, toc  
Measure performance using stopwatch timer

Timer

delete (timer)  
Remove timer object from memory

disp (timer)  
Information about timer object

get (timer)  
Timer object properties

isvalid (timer)  
Determine whether timer object is valid

set (timer)  
Configure or display timer object properties

start  
Start timer(s) running

startat  
Start timer(s) running at specified time

stop  
Stop timer(s)
Programming and Data Types

- **timer**
  - Construct timer object

- **timerfind**
  - Find timer objects

- **timerfindall**
  - Find timer objects, including invisible objects

- **wait**
  - Wait until timer stops running

Variables and Functions in Memory

- **ans**
  - Most recent answer

- **assignin**
  - Assign value to variable in specified workspace

- **datatipinfo**
  - Produce short description of input variable

- **genvarname**
  - Construct valid variable name from string

- **global**
  - Declare global variables

- **inmem**
  - Names of M-files, MEX-files, Sun Java classes in memory

- **isglobal**
  - Determine whether input is global variable

- **memory**
  - Display memory information

- **mislocked**
  - Determine whether M-file or MEX-file cannot be cleared from memory

- **mlock**
  - Prevent clearing M-file or MEX-file from memory

- **munlock**
  - Allow clearing M-file or MEX-file from memory

- **namelengthmax**
  - Maximum identifier length

- **pack**
  - Consolidate workspace memory
### Function Reference

<table>
<thead>
<tr>
<th>Persistent</th>
<th>Rehash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define persistent variable</td>
<td>Refresh function and file system path caches</td>
</tr>
</tbody>
</table>

### Control Flow

<table>
<thead>
<tr>
<th>Break</th>
<th>Terminate execution of for or while loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case</td>
<td>Execute block of code if condition is true</td>
</tr>
<tr>
<td>Catch</td>
<td>Specify how to respond to error in try statement</td>
</tr>
<tr>
<td>Continue</td>
<td>Pass control to next iteration of for or while loop</td>
</tr>
<tr>
<td>Else</td>
<td>Execute statements if condition is false</td>
</tr>
<tr>
<td>Elseif</td>
<td>Execute statements if additional condition is true</td>
</tr>
<tr>
<td>End</td>
<td>Terminate block of code, or indicate last array index</td>
</tr>
<tr>
<td>Error</td>
<td>Display message and abort function</td>
</tr>
<tr>
<td>For</td>
<td>Execute block of code specified number of times</td>
</tr>
<tr>
<td>If</td>
<td>Execute statements if condition is true</td>
</tr>
<tr>
<td>Otherwise</td>
<td>Default part of switch statement</td>
</tr>
<tr>
<td>Parfor</td>
<td>Parallel for-loop</td>
</tr>
<tr>
<td>Return</td>
<td>Return to invoking function</td>
</tr>
<tr>
<td>Switch</td>
<td>Switch among several cases, based on expression</td>
</tr>
</tbody>
</table>
try

Attempt to execute block of code, and catch errors

while

Repeatedly execute statements while condition is true

**Error Handling**

addCause (MException)

Append MException objects

assert

Generate error when condition is violated

catch

Specify how to respond to error in try statement

disp (MException)

Display MException object

eq (MException)

Compare MException objects for equality

error

Display message and abort function

ferror

Query MATLAB software about errors in file input or output

getReport (MException)

Get error message for exception

intwarning

Control state of integer warnings

isequal (MException)

Compare MException objects for equality

last (MException)

Last uncaught exception

lasterr

Last error message

lasterror

Last error message and related information

lastwarn

Last warning message

MException

Construct MException object

ne (MException)

Compare MException objects for inequality

rethrow

Reissue error
rethrow (MException)  Reissue existing exception
throw (MException)  Terminate function and issue exception
try  Attempt to execute block of code, and catch errors
warning  Warning message

**MEX Programming**

dbmex  Enable MEX-file debugging (on UNIX platforms)
inmem  Names of M-files, MEX-files, Sun Java classes in memory
mex  Compile MEX-function from C/ C++ or Fortran source code
mex.getCompilerConfigurations  Get compiler configuration information for building MEX-files
mexext  Binary MEX-file name extension
**Object-Oriented Programming**

Classes and Objects (p. 1-77)
Get information about classes and objects

Handle Classes (p. 1-78)
Define and use handle classes

Events and Listeners (p. 1-79)
Define and use events and listeners

Meta-Classes (p. 1-79)
Access information about classes without requiring instances

**Classes and Objects**

- **class**
  Create object or return class of object

- **classdef**
  Class definition key words

- **fieldnames**
  Field names of structure, or public fields of object

- **import**
  Add package or class to current import list

- **inferiorto**
  Specify inferior class relationship

- **isa**
  Determine whether input is object of given class

- **isobject**
  Determine if input is MATLAB object

- **loadobj**
  Modify how `load` function loads objects

- **methods**
  Information on class methods

- **methodsvview**
  Information on class methods in separate window

- **properties**
  Display class property names

- **saveobj**
  Modify how `save` function saves objects

- **subsasgn**
  Subscripted assignment for objects
subsindex

Subscripted indexing using object as index

subsref

Subscripted reference for objects

substruct

Create structure argument for subsasgn or subsref

superiorto

Establish superior class relationship

**Handle Classes**

addlistener (handle)

Create event listener

addprop (dynamicprops)

Add dynamic property

delete (handle)

Handle object destructor function

dynamicprops

Abstract class used to derive handle class with dynamic properties

findobj (handle)

Find objects matching specified conditions

findprop (handle)

Find meta.property object associated with property name

get (hgsetget)

Query property values of handle objects derived from hgsetget class

getdisp (hgsetget)

Override to change command window display

handle

Abstract class for deriving handle classes

hgsetget

Abstract class used to derive handle class with set and get methods

isvalid (handle)

Is object valid handle class object

notify (handle)

Notify listeners that event is occurring

relationaloperators (handle)

Equality and sorting of handle objects
Object-Oriented Programming

**set (hgsetget)**
Assign property values to handle objects derived from `hgsetget` class

**setdisp (hgsetget)**
Override to change command window display

### Events and Listeners

**addlistener (handle)**
Create event listener

**event.EventData**
Base class for all data objects passed to event listeners

**event.listener**
Class defining listener objects

**event.PropertyEvent**
Listener for property events

**event.proplistener**
Define listener object for property events

**events**
Display class event names

**notify (handle)**
Notify listeners that event is occurring

### Meta-Classes

**meta.class**
`meta.class` class describes MATLAB classes

**meta.class.fromName**
Return `meta.class` object associated with named class

**meta.DynamicProperty**
`meta.DynamicProperty` class describes dynamic property of MATLAB object

**meta.event**
`meta.event` class describes MATLAB class events

**meta.method**
`meta.method` class describes MATLAB class methods
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>meta.package</code></td>
<td><code>meta.package</code> class describes MATLAB packages</td>
</tr>
<tr>
<td><code>meta.package.fromName</code></td>
<td>Return <code>meta.package</code> object for specified package</td>
</tr>
<tr>
<td><code>meta.package.getAllPackages</code></td>
<td>Get all top-level packages</td>
</tr>
<tr>
<td><code>meta.property</code></td>
<td><code>meta.property</code> class describes MATLAB class properties</td>
</tr>
<tr>
<td><code>metaclass</code></td>
<td>Return <code>meta.class</code> object</td>
</tr>
</tbody>
</table>
# File I/O

File Name Construction (p. 1-81)  
Get path, directory, filename information; construct filenames

File Opening, Loading, and Saving (p. 1-82)  
Open files; transfer data between files and MATLAB workspace

Memory Mapping (p. 1-82)  
Access file data via memory map using MATLAB array indexing

Low-Level File I/O (p. 1-82)  
Low-level operations that use a file identifier

Text Files (p. 1-83)  
Delimited or formatted I/O to text files

XML Documents (p. 1-84)  
Documents written in Extensible Markup Language

Spreadsheets (p. 1-84)  
Excel and Lotus 1-2-3 files

Scientific Data (p. 1-85)  
CDF, FITS, HDF formats

Audio and Audio/Video (p. 1-88)  
General audio functions; SparcStation, WAVE, AVI files

Images (p. 1-90)  
Graphics files

Internet Exchange (p. 1-91)  
URL, FTP, zip, tar, and e-mail

To see a listing of file formats that are readable from MATLAB, go to file formats.

## File Name Construction

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filemarker</td>
<td>Character to separate file name and internal function name</td>
</tr>
<tr>
<td>fileparts</td>
<td>Parts of file name and path</td>
</tr>
<tr>
<td>filesep</td>
<td>Directory separator for current platform</td>
</tr>
<tr>
<td>fullfile</td>
<td>Build full filename from parts</td>
</tr>
</tbody>
</table>

1-81
tempdir
Name of system’s temporary directory

tempname
Unique name for temporary file

File Opening, Loading, and Saving

daqread
Read Data Acquisition Toolbox™ (.daq) file

importdata
Load data from disk file

load
Load workspace variables from disk

open
Open files based on extension

save
Save workspace variables to disk

uiimport
Open Import Wizard to import data

winopen
Open file in appropriate application (Windows)

Memory Mapping

disp (memmapfile)
Information about memmapfile object

get (memmapfile)
Memmapfile object properties

memmapfile
Construct memmapfile object

Low-Level File I/O

fclose
Close one or more open files

feof
Test for end-of-file

ferror
Query MATLAB software about errors in file input or output
fgetl Read line from file, discarding newline characters
fgets Read line from file, keeping newline characters
fopen Open file, or obtain information about open files
fprintf Write formatted data to file
fread Read binary data from file
rewind Move file position indicator to beginning of open file
fscanf Read formatted data from a text file
fseek Set file position indicator
ftell File position indicator
fwrite Write binary data to file

Text Files

csvread Read comma-separated value file
csvwrite Write comma-separated value file
dlmread Read ASCII-delimited file of numeric data into matrix
dlmwrite Write matrix to ASCII-delimited file
fileread Return contents of file as string vector
textread Read data from text file; write to multiple outputs
textscan Read formatted data from text file or string
XML Documents

xmlread
Parse XML document and return Document Object Model node

xmlwrite
Serialize XML Document Object Model node

xslt
Transform XML document using XSLT engine

Spreadsheets

Microsoft Excel (p. 1-84)
Read and write Microsoft Excel spreadsheet

Lotus 1-2-3 (p. 1-84)
Read and write Lotus WK1 spreadsheet

Microsoft Excel

xlsinfo
Determine whether file contains a Microsoft® Excel® spreadsheet

xlsread
Read Microsoft Excel spreadsheet file

xlsxwrite
Write Microsoft Excel spreadsheet file

Lotus 1-2-3

wk1info
Determine whether file contains 1-2-3 WK1 worksheet

wk1read
Read Lotus 1-2-3 WK1 spreadsheet file into matrix

wk1write
Write matrix to Lotus 1-2-3 WK1 spreadsheet file
## Scientific Data

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Data Format (p. 1-85)</td>
<td>Work with CDF files</td>
</tr>
<tr>
<td>Network Common Data Form (p. 1-85)</td>
<td>Work with netCDF files</td>
</tr>
<tr>
<td>Flexible Image Transport System (p. 1-87)</td>
<td>Work with FITS files</td>
</tr>
<tr>
<td>Hierarchical Data Format (p. 1-87)</td>
<td>Work with HDF files</td>
</tr>
<tr>
<td>Band-Interleaved Data (p. 1-88)</td>
<td>Work with band-interleaved files</td>
</tr>
</tbody>
</table>

## Common Data Format

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdfepoch</td>
<td>Construct cdfepoch object for Common Data Format (CDF) export</td>
</tr>
<tr>
<td>cdfinfo</td>
<td>Information about Common Data Format (CDF) file</td>
</tr>
<tr>
<td>cdfread</td>
<td>Read data from Common Data Format (CDF) file</td>
</tr>
<tr>
<td>cdfwrite</td>
<td>Write data to Common Data Format (CDF) file</td>
</tr>
<tr>
<td>todatenum</td>
<td>Convert CDF epoch object to MATLAB datenum</td>
</tr>
</tbody>
</table>

## Network Common Data Form

### File Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>netcdf</td>
<td>Summary of MATLAB Network Common Data Form (netCDF) capabilities</td>
</tr>
<tr>
<td>netcdf.abort</td>
<td>Revert recent netCDF file definitions</td>
</tr>
<tr>
<td>netcdf.close</td>
<td>Close netCDF file</td>
</tr>
<tr>
<td>netcdf.create</td>
<td>Create new netCDF dataset</td>
</tr>
</tbody>
</table>
netcdf.endDef
End netCDF file define mode

netcdf.getConstant
Return numeric value of named constant

netcdf.getConstantNames
Return list of constants known to netCDF library

netcdf.inq
Return information about netCDF file

netcdf.inqLibVers
Return netCDF library version information

netcdf.open
Open netCDF file

netcdf.reDef
Put open netCDF file into define mode

netcdf.setDefaultFormat
Change default netCDF file format

netcdf.setFill
Set netCDF fill mode

netcdf.sync
Synchronize netCDF file to disk

**Dimensions**

netcdf.defDim
Create netCDF dimension

netcdf.inqDim
Return netCDF dimension name and length

netcdf.inqDimID
Return dimension ID

netcdf.renameDim
Change name of netCDF dimension

**Variables**

netcdf.defVar
Create netCDF variable

netcdf.getVar
Return data from netCDF variable

netcdf.inqVar
Return information about variable

netcdf.inqVarID
Return ID associated with variable name
netcdf.putVar
Write data to netCDF variable
netcdf.renameVar
Change name of netCDF variable

Attributes
netcdf.copyAtt
Copy attribute to new location
netcdf.delAtt
Delete netCDF attribute
netcdf.getAtt
Return netCDF attribute
netcdf.inqAtt
Return information about netCDF attribute
netcdf.inqAttID
Return ID of netCDF attribute
netcdf.inqAttName
Return name of netCDF attribute
netcdf.putAtt
Write netCDF attribute
netcdf.renameAtt
Change name of attribute

Flexible Image Transport System
fitsinfo
Information about FITS file
fitsread
Read data from FITS file

Hierarchical Data Format
hdf
Summary of MATLAB HDF4 capabilities
hdf5
Summary of MATLAB HDF5 capabilities
hdf5info
Information about HDF5 file
hdf5read
Read HDF5 file
hdf5write
Write data to file in HDF5 format
hdfinfo
Information about HDF4 or HDF-EOS file

hdfread
Read data from HDF4 or HDF-EOS file

hdftool
Browse and import data from HDF4 or HDF-EOS files

**Band-Interleaved Data**

multibandread
Read band-interleaved data from binary file

multibandwrite
Write band-interleaved data to file

**Audio and Audio/Video**

Utilities (p. 1-88)
Create audio player object, obtain information about multimedia files, convert to/from audio signal

SPARCstation-Specific Sound (p. 1-89)
Access NeXT/SUN (.au) sound files

Microsoft WAVE Sound (p. 1-89)
Access Microsoft WAVE (.wav) sound files

Audio/Video Interleaved (p. 1-90)
Access Audio/Video interleaved (.avi) sound files

**Utilities**

audiodevinfo
Information about audio device

audioplayer
Create audioplayer object

audiorecorder
Create audiorecorder object

beep
Produce beep sound
lin2mu  Convert linear audio signal to mu-law
mmfileinfo  Information about multimedia file
mmreader  Create multimedia reader object for reading video files
mmreader.isPlatformSupported  Determine whether mmreader function is available on current platform
mu2lin  Convert mu-law audio signal to linear
read  Read video frame data from multimedia reader object
sound  Convert vector into sound
soundsc  Scale data and play as sound

**SPARCstation-Specific Sound**

aufinfo  Information about NeXT/SUN (.au) sound file
auread  Read NeXT/SUN (.au) sound file
auwrite  Write NeXT/SUN (.au) sound file

**Microsoft WAVE Sound**

wavinfo  Information about Microsoft WAVE (.wav) sound file
wavplay  Play recorded sound on PC-based audio output device
wavread  Read Microsoft WAVE (.wav) sound file
### Function Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wavrecord</td>
<td>Record sound using PC-based audio input device</td>
</tr>
<tr>
<td>wavwrite</td>
<td>Write Microsoft WAVE (.wav) sound file</td>
</tr>
<tr>
<td><strong>Audio/Video Interleaved</strong></td>
<td></td>
</tr>
<tr>
<td>addframe</td>
<td>Add frame to Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>avifile</td>
<td>Create new Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>aviinfo</td>
<td>Information about Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>aviread</td>
<td>Read Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>close (avifile)</td>
<td>Close Audio/Video Interleaved (AVI) file</td>
</tr>
<tr>
<td>movie2avi</td>
<td>Create Audio/Video Interleaved (AVI) movie from MATLAB movie</td>
</tr>
</tbody>
</table>

### Images

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exifread</td>
<td>Read EXIF information from JPEG and TIFF image files</td>
</tr>
<tr>
<td>im2java</td>
<td>Convert image to Java image</td>
</tr>
<tr>
<td>imfinfo</td>
<td>Information about graphics file</td>
</tr>
<tr>
<td>imread</td>
<td>Read image from graphics file</td>
</tr>
<tr>
<td>imwrite</td>
<td>Write image to graphics file</td>
</tr>
</tbody>
</table>
Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-91)
Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files

FTP (p. 1-91)
Connect to FTP server, download from server, manage FTP files, close server connection

URL, Zip, Tar, E-Mail

gunzip
Uncompress GNU zip files
gzip
Compress files into GNU zip files
sendmail
Send e-mail message to address list
tar
Compress files into tar file
untar
Extract contents of tar file
unzip
Extract contents of zip file
urlread
Read content at URL
urlwrite
Save contents of URL to file
zip
Compress files into zip file

FTP

ascii
Set FTP transfer type to ASCII
binary
Set FTP transfer type to binary
cd (ftp)
Change current directory on FTP server
close (ftp)
Close connection to FTP server
delete (ftp)
Remove file on FTP server
dir (ftp)
Directory contents on FTP server
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp</td>
<td>Connect to FTP server, creating FTP object</td>
</tr>
<tr>
<td>mget</td>
<td>Download file from FTP server</td>
</tr>
<tr>
<td>mkdir (ftp)</td>
<td>Create new directory on FTP server</td>
</tr>
<tr>
<td>mput</td>
<td>Upload file or directory to FTP server</td>
</tr>
<tr>
<td>rename</td>
<td>Rename file on FTP server</td>
</tr>
<tr>
<td>rmdir (ftp)</td>
<td>Remove directory on FTP server</td>
</tr>
</tbody>
</table>
# Graphics

<table>
<thead>
<tr>
<th>Basic Plots and Graphs (p. 1-93)</th>
<th>Linear line plots, log and semilog plots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plotting Tools (p. 1-94)</td>
<td>GUIs for interacting with plots</td>
</tr>
<tr>
<td>Annotating Plots (p. 1-94)</td>
<td>Functions for and properties of titles, axes labels, legends, mathematical symbols</td>
</tr>
<tr>
<td>Specialized Plotting (p. 1-95)</td>
<td>Bar graphs, histograms, pie charts, contour plots, function plotters</td>
</tr>
<tr>
<td>Bit-Mapped Images (p. 1-99)</td>
<td>Display image object, read and write graphics file, convert to movie frames</td>
</tr>
<tr>
<td>Printing (p. 1-99)</td>
<td>Printing and exporting figures to standard formats</td>
</tr>
<tr>
<td>Handle Graphics (p. 1-100)</td>
<td>Creating graphics objects, setting properties, finding handles</td>
</tr>
</tbody>
</table>

## Basic Plots and Graphs

<table>
<thead>
<tr>
<th>box</th>
<th>Axes border</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorbar</td>
<td>Plot error bars along curve</td>
</tr>
<tr>
<td>hold</td>
<td>Retain current graph in figure</td>
</tr>
<tr>
<td>LineSpec (Line Specification)</td>
<td>Line specification string syntax</td>
</tr>
<tr>
<td>loglog</td>
<td>Log-log scale plot</td>
</tr>
<tr>
<td>plot</td>
<td>2-D line plot</td>
</tr>
<tr>
<td>plot3</td>
<td>3-D line plot</td>
</tr>
<tr>
<td>plotyy</td>
<td>2-D line plots with y-axes on both left and right side</td>
</tr>
<tr>
<td>polar</td>
<td>Polar coordinate plot</td>
</tr>
<tr>
<td>semilogx, semilogy</td>
<td>Semilogarithmic plots</td>
</tr>
<tr>
<td>subplot</td>
<td>Create axes in tiled positions</td>
</tr>
</tbody>
</table>
Plotting Tools

figurepalette
pan
plotbrowser
plotedit
plottools
propertyeditor
rotate3d
showplottool
zoom

Show or hide figure palette
Pan view of graph interactively
Show or hide figure plot browser
Interactively edit and annotate plots
Show or hide plot tools
Show or hide property editor
Rotate 3-D view using mouse
Show or hide figure plot tool
Turn zooming on or off or magnify by factor

Annotating Plots

annotation
clabel
datacursormode
datetick
gtext
legend
line
rectangle
texlabel
title
xlabel, ylabel, zlabel

Create annotation objects
Contour plot elevation labels
Enable or disable interactive data cursor mode
Date formatted tick labels
Mouse placement of text in 2-D view
Graph legend for lines and patches
Create line object
Create 2-D rectangle object
Produce TeX format from character string
Add title to current axes
Label x-, y-, and z-axis
Specialized Plotting

Area, Bar, and Pie Plots (p. 1-95) 1-D, 2-D, and 3-D graphs and charts
Contour Plots (p. 1-96) Unfilled and filled contours in 2-D and 3-D
Direction and Velocity Plots (p. 1-96) Comet, compass, feather and quiver plots
Discrete Data Plots (p. 1-96) Stair, step, and stem plots
Function Plots (p. 1-96) Easy-to-use plotting utilities for graphing functions
Histograms (p. 1-97) Plots for showing distributions of data
Polygons and Surfaces (p. 1-97) Functions to generate and plot surface patches in two or more dimensions
Scatter/Bubble Plots (p. 1-98) Plots of point distributions
Animation (p. 1-98) Functions to create and play movies of plots

Area, Bar, and Pie Plots

area Filled area 2-D plot
bar, barh Plot bar graph (vertical and horizontal)
bar3, bar3h Plot 3-D bar chart
pareto Pareto chart
pie Pie chart
pie3 3-D pie chart
**Contour Plots**

- `contour` - Contour plot of matrix
- `contour3` - 3-D contour plot
- `contourc` - Low-level contour plot computation
- `contourf` - Filled 2-D contour plot
- `ezcontour` - Easy-to-use contour plotter
- `ezcontourf` - Easy-to-use filled contour plotter

**Direction and Velocity Plots**

- `comet` - 2-D comet plot
- `comet3` - 3-D comet plot
- `compass` - Plot arrows emanating from origin
- `feather` - Plot velocity vectors
- `quiver` - Quiver or velocity plot
- `quiver3` - 3-D quiver or velocity plot

**Discrete Data Plots**

- `stairs` - Stairstep graph
- `stem` - Plot discrete sequence data
- `stem3` - Plot 3-D discrete sequence data

**Function Plots**

- `ezcontour` - Easy-to-use contour plotter
- `ezcontourf` - Easy-to-use filled contour plotter
- `ezmesh` - Easy-to-use 3-D mesh plotter
### Graphics

- **ezmeshc**
  - Easy-to-use combination mesh/contour plotter
- **ezplot**
  - Easy-to-use function plotter
- **ezplot3**
  - Easy-to-use 3-D parametric curve plotter
- **ezpolar**
  - Easy-to-use polar coordinate plotter
- **ezsurf**
  - Easy-to-use 3-D colored surface plotter
- **ezsurfc**
  - Easy-to-use combination surface/contour plotter
- **fplot**
  - Plot function between specified limits

### Histograms

- **hist**
  - Histogram plot
- **histc**
  - Histogram count
- **rose**
  - Angle histogram plot

### Polygons and Surfaces

- **convhull**
  - Convex hull
- **cylinder**
  - Generate cylinder
- **delaunay**
  - Delaunay triangulation
- **delaunay3**
  - 3-D Delaunay tessellation
- **delaunayn**
  - N-D Delaunay tessellation
- **dsearch**
  - Search Delaunay triangulation for nearest point
- **dsearchn**
  - N-D nearest point search
- **ellipsoid**
  - Generate ellipsoid
fill
fill3
inpoly
pcolor
polyarea
rectint
ribbon
slice
sphere
tsearch
tsearchn
voronoi
waterfall

**Scatter/Bubble Plots**

plotmatrix
scatter
scatter3

**Animation**

frame2im
getframe
im2frame

Filled 2-D polygons
Filled 3-D polygons
Points inside polygonal region
Pseudocolor (checkerboard) plot
Area of polygon
Rectangle intersection area
Ribbon plot
Volumetric slice plot
Generate sphere
Search for enclosing Delaunay triangle
N-D closest simplex search
Voronoi diagram
Waterfall plot

Scatter plot matrix
Scatter plot
3-D scatter plot

Return image data associated with movie frame
Capture movie frame
Convert image to movie frame
movie
noanimate

**Bit-Mapped Images**

frame2im
im2frame
im2java
image
imagesc
imfinfo
imformats
imread
imwrite
ind2rgb

Return image data associated with movie frame
Convert image to movie frame
Convert image to Java image
Display image object
Scale data and display image object
Information about graphics file
Manage image file format registry
Read image from graphics file
Write image to graphics file
Convert indexed image to RGB image

**Printing**

hgexport
orient
print, printopt
printdlg
printpreview
saveas

Export figure
Hardcopy paper orientation
Print figure or save to file and configure printer defaults
Print dialog box
Preview figure to print
Save figure or Simulink block diagram using specified format
**Handle Graphics**

Graphics Object Identification (p. 1-100)  
Find and manipulate graphics objects via their handles

Object Creation (p. 1-101)  
 Constructors for core graphics objects

Plot Objects (p. 1-101)  
Property descriptions for plot objects

Figure Windows (p. 1-102)  
Control and save figures

Axes Operations (p. 1-103)  
Operate on axes objects

Object Property Operations (p. 1-103)  
Query, set, and link object properties

**Graphics Object Identification**

- `allchild`  
  Find all children of specified objects

- `ancestor`  
  Ancestor of graphics object

- `copyobj`  
  Copy graphics objects and their descendants

- `delete`  
  Remove files or graphics objects

- `findall`  
  Find all graphics objects

- `findfigs`  
  Find visible offscreen figures

- `findobj`  
  Locate graphics objects with specific properties

- `gca`  
  Current axes handle

- `gcbf`  
  Handle of figure containing object whose callback is executing

- `gcbo`  
  Handle of object whose callback is executing

- `gco`  
  Handle of current object

- `get`  
  Query Handle Graphics® object properties
ishandle  Determine whether input is valid
          Handle Graphics handle
propedit  Open Property Editor
set       Set Handle Graphics object
          properties

### Object Creation

axes       Create axes graphics object
figure     Create figure graphics object
hggroup    Create hggroup object
hgtransform Create hgtransform graphics object
image      Display image object
light      Create light object
line       Create line object
patch      Create patch graphics object
rectangle  Create 2-D rectangle object
rootobject Root
surface    Create surface object
text       Create text object in current axes
uicontextmenu Create context menu

### Plot Objects

Annotation Arrow Properties Define annotation arrow properties
Annotation Doublearrow Properties Define annotation doublearrow properties
Annotation Ellipse Properties Define annotation ellipse properties
Annotation Line Properties Define annotation line properties
### Figure Windows

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clf</td>
<td>Clear current figure window</td>
</tr>
<tr>
<td>close</td>
<td>Remove specified figure</td>
</tr>
<tr>
<td>closereq</td>
<td>Default figure close request function</td>
</tr>
<tr>
<td>drawnow</td>
<td>Flush event queue and update figure window</td>
</tr>
<tr>
<td>gcf</td>
<td>Current figure handle</td>
</tr>
<tr>
<td>hgload</td>
<td>Load Handle Graphics object hierarchy from file</td>
</tr>
<tr>
<td>hgsave</td>
<td>Save Handle Graphics object hierarchy to file</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>newplot</td>
<td>Determine where to draw graphics objects</td>
</tr>
<tr>
<td>opengl</td>
<td>Control OpenGL® rendering</td>
</tr>
<tr>
<td>refresh</td>
<td>Redraw current figure</td>
</tr>
<tr>
<td>saveas</td>
<td>Save figure or Simulink block diagram using specified format</td>
</tr>
</tbody>
</table>

**Axes Operations**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axis</td>
<td>Axis scaling and appearance</td>
</tr>
<tr>
<td>box</td>
<td>Axes border</td>
</tr>
<tr>
<td>cla</td>
<td>Clear current axes</td>
</tr>
<tr>
<td>gca</td>
<td>Current axes handle</td>
</tr>
<tr>
<td>grid</td>
<td>Grid lines for 2-D and 3-D plots</td>
</tr>
<tr>
<td>ishold</td>
<td>Current hold state</td>
</tr>
<tr>
<td>makehgtform</td>
<td>Create 4-by-4 transform matrix</td>
</tr>
</tbody>
</table>

**Object Property Operations**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>Query Handle Graphics object properties</td>
</tr>
<tr>
<td>linkaxes</td>
<td>Synchronize limits of specified 2-D axes</td>
</tr>
<tr>
<td>linkprop</td>
<td>Keep same value for corresponding properties</td>
</tr>
<tr>
<td>refreshdata</td>
<td>Refresh data in graph when data source is specified</td>
</tr>
<tr>
<td>set</td>
<td>Set Handle Graphics object properties</td>
</tr>
</tbody>
</table>
# 3-D Visualization

**Surface and Mesh Plots** (p. 1-104)  
Plot matrices, visualize functions of two variables, specify colormap  

**View Control** (p. 1-106)  
Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits  

**Lighting** (p. 1-108)  
Add and control scene lighting  

**Transparency** (p. 1-108)  
Specify and control object transparency  

**Volume Visualization** (p. 1-108)  
Visualize gridded volume data  

## Surface and Mesh Plots

**Surface and Mesh Creation** (p. 1-104)  
Visualizing gridded and triangulated data as lines and surfaces  

**Domain Generation** (p. 1-105)  
Gridding data and creating arrays  

**Color Operations** (p. 1-105)  
Specifying, converting, and manipulating color spaces, colormaps, colorbars, and backgrounds  

## Surface and Mesh Creation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hidden</td>
<td>Remove hidden lines from mesh plot</td>
</tr>
<tr>
<td>mesh, meshc, meshz</td>
<td>Mesh plots</td>
</tr>
<tr>
<td>peaks</td>
<td>Example function of two variables</td>
</tr>
<tr>
<td>surf, surfc</td>
<td>3-D shaded surface plot</td>
</tr>
<tr>
<td>surface</td>
<td>Create surface object</td>
</tr>
<tr>
<td>surfl</td>
<td>Surface plot with colormap-based lighting</td>
</tr>
<tr>
<td>tetramesh</td>
<td>Tetrahedron mesh plot</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>trimesh</td>
<td>Triangular mesh plot</td>
</tr>
<tr>
<td>triplot</td>
<td>2-D triangular plot</td>
</tr>
<tr>
<td>trisurf</td>
<td>Triangular surface plot</td>
</tr>
</tbody>
</table>

### Domain Generation

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>griddata</td>
<td>Data gridding</td>
</tr>
<tr>
<td>meshgrid</td>
<td>Generate X and Y arrays for 3-D plots</td>
</tr>
</tbody>
</table>

### Color Operations

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brighten</td>
<td>Brighten or darken colormap</td>
</tr>
<tr>
<td>caxis</td>
<td>Color axis scaling</td>
</tr>
<tr>
<td>colorbar</td>
<td>Colorbar showing color scale</td>
</tr>
<tr>
<td>colordef</td>
<td>Set default property values to display different color schemes</td>
</tr>
<tr>
<td>colormap</td>
<td>Set and get current colormap</td>
</tr>
<tr>
<td>colormapeditor</td>
<td>Start colormap editor</td>
</tr>
<tr>
<td>ColorSpec (Color Specification)</td>
<td>Color specification</td>
</tr>
<tr>
<td>contrast</td>
<td>Grayscale colormap for contrast enhancement</td>
</tr>
<tr>
<td>graymon</td>
<td>Set default figure properties for grayscale monitors</td>
</tr>
<tr>
<td>hsv2rgb</td>
<td>Convert HSV colormap to RGB colormap</td>
</tr>
<tr>
<td>rgb2hsv</td>
<td>Convert RGB colormap to HSV colormap</td>
</tr>
<tr>
<td>rgbplot</td>
<td>Plot colormap</td>
</tr>
<tr>
<td>shading</td>
<td>Set color shading properties</td>
</tr>
<tr>
<td>spinmap</td>
<td>Spin colormap</td>
</tr>
</tbody>
</table>
surfnorm
  Compute and display 3-D surface normals

whitebg
  Change axes background color

**View Control**

Camera Viewpoint (p. 1-106)
  Orbiting, dollying, pointing, rotating camera positions and setting fields of view

Aspect Ratio and Axis Limits (p. 1-107)
  Specifying what portions of axes to view and how to scale them

Object Manipulation (p. 1-107)
  Panning, rotating, and zooming views

Region of Interest (p. 1-107)
  Interactively identifying rectangular regions

**Camera Viewpoint**

camdolly
  Move camera position and target

cameratoolbar
  Control camera toolbar programmatically

camlookat
  Position camera to view object or group of objects

camorbit
  Rotate camera position around camera target

campan
  Rotate camera target around camera position

campos
  Set or query camera position

camproj
  Set or query projection type

camroll
  Rotate camera about view axis

camtargert
  Set or query location of camera target
camup
Set or query camera up vector

camva
Set or query camera view angle

camzoom
Zoom in and out on scene

makehtform
Create 4-by-4 transform matrix

view
Viewpoint specification

viewmtx
View transformation matrices

Aspect Ratio and Axis Limits

daspect
Set or query axes data aspect ratio

pbaspect
Set or query plot box aspect ratio

xlim, ylim, zlim
Set or query axis limits

Object Manipulation

pan
Pan view of graph interactively

reset
Reset graphics object properties to their defaults

rotate
Rotate object in specified direction

rotate3d
Rotate 3-D view using mouse

selectmoveresize
Select, move, resize, or copy axes and uicontrol graphics objects

zoom
Turn zooming on or off or magnify by factor

Region of Interest

dragrect
Drag rectangles with mouse

rbbox
Create rubberband box for area selection
## Lighting

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>camlight</td>
<td>Create or move light object in camera coordinates</td>
</tr>
<tr>
<td>diffuse</td>
<td>Calculate diffuse reflectance</td>
</tr>
<tr>
<td>light</td>
<td>Create light object</td>
</tr>
<tr>
<td>lightangle</td>
<td>Create or position <code>light</code> object in spherical coordinates</td>
</tr>
<tr>
<td>lighting</td>
<td>Specify lighting algorithm</td>
</tr>
<tr>
<td>material</td>
<td>Control reflectance properties of surfaces and patches</td>
</tr>
<tr>
<td>specular</td>
<td>Calculate specular reflectance</td>
</tr>
</tbody>
</table>

## Transparency

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alim</td>
<td>Set or query axes alpha limits</td>
</tr>
<tr>
<td>alpha</td>
<td>Set transparency properties for objects in current axes</td>
</tr>
<tr>
<td>alphamap</td>
<td>Specify figure alphamap (transparency)</td>
</tr>
</tbody>
</table>

## Volume Visualization

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coneplot</td>
<td>Plot velocity vectors as cones in 3-D vector field</td>
</tr>
<tr>
<td>contourslice</td>
<td>Draw contours in volume slice planes</td>
</tr>
<tr>
<td>curl</td>
<td>Compute curl and angular velocity of vector field</td>
</tr>
<tr>
<td>divergence</td>
<td>Compute divergence of vector field</td>
</tr>
<tr>
<td>flow</td>
<td>Simple function of three variables</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>interpstreamspeed</td>
<td>Interpolate stream-line vertices from flow speed</td>
</tr>
<tr>
<td>isocaps</td>
<td>Compute isosurface end-cap geometry</td>
</tr>
<tr>
<td>isocolors</td>
<td>Calculate isosurface and patch colors</td>
</tr>
<tr>
<td>isonormals</td>
<td>Compute normals of isosurface vertices</td>
</tr>
<tr>
<td>isosurface</td>
<td>Extract isosurface data from volume data</td>
</tr>
<tr>
<td>reducepatch</td>
<td>Reduce number of patch faces</td>
</tr>
<tr>
<td>reducevolume</td>
<td>Reduce number of elements in volume data set</td>
</tr>
<tr>
<td>shrinkfaces</td>
<td>Reduce size of patch faces</td>
</tr>
<tr>
<td>slice</td>
<td>Volumetric slice plot</td>
</tr>
<tr>
<td>smooth3</td>
<td>Smooth 3-D data</td>
</tr>
<tr>
<td>stream2</td>
<td>Compute 2-D streamline data</td>
</tr>
<tr>
<td>stream3</td>
<td>Compute 3-D streamline data</td>
</tr>
<tr>
<td>streamline</td>
<td>Plot streamlines from 2-D or 3-D vector data</td>
</tr>
<tr>
<td>streamparticles</td>
<td>Plot stream particles</td>
</tr>
<tr>
<td>streamribbon</td>
<td>3-D stream ribbon plot from vector volume data</td>
</tr>
<tr>
<td>streamslice</td>
<td>Plot streamlines in slice planes</td>
</tr>
<tr>
<td>streamtube</td>
<td>Create 3-D stream tube plot</td>
</tr>
<tr>
<td>subvolume</td>
<td>Extract subset of volume data set</td>
</tr>
<tr>
<td>surf2patch</td>
<td>Convert surface data to patch data</td>
</tr>
<tr>
<td>volumebounds</td>
<td>Coordinate and color limits for volume data</td>
</tr>
</tbody>
</table>
**GUI Development**

Predefined Dialog Boxes (p. 1-110)  
Dialog boxes for error, user input, waiting, etc.

User Interface Deployment (p. 1-111)  
Open GUIs, create the handles structure

User Interface Development (p. 1-111)  
Start GUIDE, manage application data, get user input

User Interface Objects (p. 1-112)  
Create GUI components

Objects from Callbacks (p. 1-113)  
Find object handles from within callbacks functions

GUI Utilities (p. 1-113)  
Move objects, wrap text

Program Execution (p. 1-114)  
Wait and resume based on user input

---

**Predefined Dialog Boxes**

dialog  
Create and display empty dialog box

erroldlg  
Create and open error dialog box

export2wsdlg  
Export variables to workspace

helpdlg  
Create and open help dialog box

inputdlg  
Create and open input dialog box

listdlg  
Create and open list-selection dialog box

msgbox  
Create and open message box

printdlg  
Print dialog box

printpreview  
Preview figure to print

questdlg  
Create and open question dialog box

uigetdir  
Open standard dialog box for selecting directory
### GUI Development

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uigetfile</td>
<td>Open standard dialog box for retrieving files</td>
</tr>
<tr>
<td>uigetpref</td>
<td>Open dialog box for retrieving preferences</td>
</tr>
<tr>
<td>uiopen</td>
<td>Open file selection dialog box with appropriate file filters</td>
</tr>
<tr>
<td>uiputfile</td>
<td>Open standard dialog box for saving files</td>
</tr>
<tr>
<td>uisave</td>
<td>Open standard dialog box for saving workspace variables</td>
</tr>
<tr>
<td>uisetcolor</td>
<td>Open standard dialog box for setting object’s ColorSpec</td>
</tr>
<tr>
<td>uisetfont</td>
<td>Open standard dialog box for setting object’s font characteristics</td>
</tr>
<tr>
<td>waitbar</td>
<td>Open or update a wait bar dialog box</td>
</tr>
<tr>
<td>warndlg</td>
<td>Open warning dialog box</td>
</tr>
</tbody>
</table>

### User Interface Deployment

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>guidata</td>
<td>Store or retrieve GUI data</td>
</tr>
<tr>
<td>guihandles</td>
<td>Create structure of handles</td>
</tr>
<tr>
<td>movegui</td>
<td>Move GUI figure to specified location on screen</td>
</tr>
<tr>
<td>openfig</td>
<td>Open new copy or raise existing copy of saved figure</td>
</tr>
</tbody>
</table>

### User Interface Development

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addpref</td>
<td>Add preference</td>
</tr>
<tr>
<td>getappdata</td>
<td>Value of application-defined data</td>
</tr>
<tr>
<td>getpref</td>
<td>Preference</td>
</tr>
</tbody>
</table>
Function Reference

**ginput**
Graphical input from mouse or cursor

**guidata**
Store or retrieve GUI data

**guide**
Open GUI Layout Editor

**inspect**
Open Property Inspector

**isappdata**
True if application-defined data exists

**ispref**
Test for existence of preference

**rmappdata**
Remove application-defined data

**rmpref**
Remove preference

**setappdata**
Specify application-defined data

**setpref**
Set preference

**uigetpref**
Open dialog box for retrieving preferences

**uisetpref**
Manage preferences used in uigetpref

**waitfor**
Wait for condition before resuming execution

**waitforbuttonpress**
Wait for key press or mouse-button click

### User Interface Objects

**menu**
Generate menu of choices for user input

**uibuttongroup**
Create container object to exclusively manage radio buttons and toggle buttons

**uicontextmenu**
Create context menu

**uicontrol**
Create user interface control object
# GUI Development

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uimenu</td>
<td>Create menus on figure windows</td>
</tr>
<tr>
<td>uipanel</td>
<td>Create panel container object</td>
</tr>
<tr>
<td>uipushtool</td>
<td>Create push button on toolbar</td>
</tr>
<tr>
<td>uitable</td>
<td>Create 2-D graphic table GUI component</td>
</tr>
<tr>
<td>uitoggletool</td>
<td>Create toggle button on toolbar</td>
</tr>
<tr>
<td>uitoolbar</td>
<td>Create toolbar on figure</td>
</tr>
</tbody>
</table>

## Objects from Callbacks

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findall</td>
<td>Find all graphics objects</td>
</tr>
<tr>
<td>findfigs</td>
<td>Find visible offscreen figures</td>
</tr>
<tr>
<td>findobj</td>
<td>Locate graphics objects with specific properties</td>
</tr>
<tr>
<td>gcbf</td>
<td>Handle of figure containing object whose callback is executing</td>
</tr>
<tr>
<td>gcbo</td>
<td>Handle of object whose callback is executing</td>
</tr>
</tbody>
</table>

## GUI Utilities

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpixelposition</td>
<td>Get component position in pixels</td>
</tr>
<tr>
<td>listfonts</td>
<td>List available system fonts</td>
</tr>
<tr>
<td>selectmoveresize</td>
<td>Select, move, resize, or copy axes and uicontrol graphics objects</td>
</tr>
<tr>
<td>setpixelposition</td>
<td>Set component position in pixels</td>
</tr>
<tr>
<td>textwrap</td>
<td>Wrapped string matrix for given uicontrol</td>
</tr>
<tr>
<td>uistack</td>
<td>Reorder visual stacking order of objects</td>
</tr>
</tbody>
</table>
## Program Execution

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uiresume</td>
<td>Resume execution of blocked M-file</td>
</tr>
<tr>
<td>uiwait</td>
<td>Block execution and wait for resume</td>
</tr>
</tbody>
</table>
External Interfaces

Shared Libraries (p. 1-115)

- Access functions stored in external shared library files

Java (p. 1-116)

- Work with objects constructed from Java API and third-party class packages

.NET (p. 1-117)

- Work with objects constructed from .NET assemblies

Component Object Model and ActiveX (p. 1-117)

- Integrate COM components into your application

Web Services (p. 1-120)

- Communicate between applications over a network using SOAP and WSDL

Serial Port Devices (p. 1-120)

- Read and write to devices connected to your computer’s serial port

See also MATLAB C and Fortran API Reference for functions you can use in external routines that interact with MATLAB programs and the data in MATLAB workspaces.

Shared Libraries

calllib

Call function in shared library

libfunctions

Return information on functions in shared library

libfunctionsview

View functions in shared library

libisloaded

Determine if shared library is loaded

libpointer

Create pointer object for use with shared libraries

libstruct

Create structure pointer for use with shared libraries
loadlibrary  Load shared library into MATLAB software
unloadlibrary Unload shared library from memory

**Java**

class  Create object or return class of object
fieldnames  Field names of structure, or public fields of object
import  Add package or class to current import list
inspect  Open Property Inspector
isa  Determine whether input is object of given class
isjava  Determine whether input is Sun Java object
javaaddpath Add entries to dynamic Sun Java class path
javaArray  Construct Sun Java array
javachck  Generate error message based on Sun Java feature support
javaclasspath Get and set Sun Java class path
javaMethod  Call Sun Java method
javaMethodEDT  Call Sun Java method from Event Dispatch Thread (EDT)
javaObject  Construct Sun Java object
javaObjectEDT  Construct Sun Java object on Event Dispatch Thread (EDT)
javarmpath Remove entries from dynamic Sun Java class path
methods  Information on class methods
### External Interfaces

- **methodsvie**w
  - Information on class methods in separate window
- **usejava**
  - Determine whether Sun Java feature is supported in MATLAB software

### .NET

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NET.addAssembly</td>
<td>Make .NET assembly visible to MATLAB</td>
</tr>
<tr>
<td>NET.convertArray</td>
<td>Convert MATLAB array to .NET array</td>
</tr>
<tr>
<td>NET.createArray</td>
<td>Create single or multidimensional .NET array</td>
</tr>
<tr>
<td>NET.createGeneric</td>
<td>Create instance of specialized .NET generic type</td>
</tr>
<tr>
<td>NET.GenericClass</td>
<td>Represent parameterized generic type definitions</td>
</tr>
<tr>
<td>NET.GenericClass Constructor</td>
<td>Constructor for NET.GenericClass class</td>
</tr>
</tbody>
</table>

### Component Object Model and ActiveX

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actxcontrol</td>
<td>Create Microsoft® ActiveX® control in figure window</td>
</tr>
<tr>
<td>actxcontrollist</td>
<td>List all currently installed Microsoft ActiveX controls</td>
</tr>
<tr>
<td>actxcontrolselect</td>
<td>Open GUI to create Microsoft ActiveX control</td>
</tr>
<tr>
<td>actxGetRunningServer</td>
<td>Get handle to running instance of Automation server</td>
</tr>
<tr>
<td>actxserver</td>
<td>Create COM server</td>
</tr>
<tr>
<td>addproperty</td>
<td>Add custom property to COM object</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>class</code></td>
<td>Create object or return class of object</td>
</tr>
<tr>
<td><code>delete (COM)</code></td>
<td>Remove COM control or server</td>
</tr>
<tr>
<td><code>deleteproperty</code></td>
<td>Remove custom property from COM object</td>
</tr>
<tr>
<td><code>enableservice</code></td>
<td>Enable, disable, or report status of Automation server</td>
</tr>
<tr>
<td><code>eventlisteners</code></td>
<td>List all event handler functions registered for COM object</td>
</tr>
<tr>
<td><code>events (COM)</code></td>
<td>List of events COM object can trigger</td>
</tr>
<tr>
<td><code>Execute</code></td>
<td>Execute MATLAB command in Automation server</td>
</tr>
<tr>
<td><code>Feval (COM)</code></td>
<td>Evaluate MATLAB function in Automation server</td>
</tr>
<tr>
<td><code>fieldnames</code></td>
<td>Field names of structure, or public fields of object</td>
</tr>
<tr>
<td><code>get (COM)</code></td>
<td>Get property value from interface, or display properties</td>
</tr>
<tr>
<td><code>GetCharArray</code></td>
<td>Get character array from Automation server</td>
</tr>
<tr>
<td><code>GetFullMatrix</code></td>
<td>Get matrix from Automation server</td>
</tr>
<tr>
<td><code>GetVariable</code></td>
<td>Get data from variable in Automation server workspace</td>
</tr>
<tr>
<td><code>GetWorkspaceData</code></td>
<td>Get data from Automation server workspace</td>
</tr>
<tr>
<td><code>inspect</code></td>
<td>Open Property Inspector</td>
</tr>
<tr>
<td><code>interfaces</code></td>
<td>List custom interfaces to COM server</td>
</tr>
<tr>
<td><code>invoke</code></td>
<td>Invoke method on COM object or interface, or display methods</td>
</tr>
<tr>
<td><code>isa</code></td>
<td>Determine whether input is object of given class</td>
</tr>
<tr>
<td><code>iscom</code></td>
<td>Is input COM object</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>isevent</code></td>
<td>True if COM object event</td>
</tr>
<tr>
<td><code>isinterface</code></td>
<td>Is input COM interface</td>
</tr>
<tr>
<td><code>ismethod</code></td>
<td>Determine whether input is COM object method</td>
</tr>
<tr>
<td><code>isprop</code></td>
<td>Determine whether input is COM object property</td>
</tr>
<tr>
<td><code>load (COM)</code></td>
<td>Initialize control object from file</td>
</tr>
<tr>
<td><code>MaximizeCommandWindow</code></td>
<td>Open Automation server window</td>
</tr>
<tr>
<td><code>methods</code></td>
<td>Information on class methods</td>
</tr>
<tr>
<td><code>methodsviw</code></td>
<td>Information on class methods in separate window</td>
</tr>
<tr>
<td><code>MinimizeCommandWindow</code></td>
<td>Minimize size of Automation server window</td>
</tr>
<tr>
<td><code>move</code></td>
<td>Move or resize control in parent window</td>
</tr>
<tr>
<td><code>propedit (COM)</code></td>
<td>Open built-in property page for control</td>
</tr>
<tr>
<td><code>PutCharArray</code></td>
<td>Store character array in Automation server</td>
</tr>
<tr>
<td><code>PutFullMatrix</code></td>
<td>Store matrix in Automation server</td>
</tr>
<tr>
<td><code>PutWorkspaceData</code></td>
<td>Store data in Automation server workspace</td>
</tr>
<tr>
<td><code>Quit (COM)</code></td>
<td>Terminate MATLAB Automation server</td>
</tr>
<tr>
<td><code>registerevent</code></td>
<td>Register event handler for COM object event at run-time</td>
</tr>
<tr>
<td><code>release</code></td>
<td>Release COM interface</td>
</tr>
<tr>
<td><code>save (COM)</code></td>
<td>Serialize control object to file</td>
</tr>
<tr>
<td><code>set (COM)</code></td>
<td>Set object or interface property to specified value</td>
</tr>
</tbody>
</table>
unregisterallevents Unregister all event handlers for COM object event at run-time
unregisterevent Unregister event handler for COM object event at run-time

**Web Services**
callSoapService Send SOAP message to endpoint
createClassFromWsd1 Create MATLAB class based on WSDL document
createSoapMessage Create SOAP message to send to server
parseSoapResponse Convert response string from SOAP server into MATLAB types

**Serial Port Devices**
instrcallback Event information when event occurs
instrfind Read serial port objects from memory to MATLAB workspace
instrfindall Find visible and hidden serial port objects
readasync Read data asynchronously from device
record Record data and event information to file
serial Create serial port object
serial.clear Remove serial port object from MATLAB workspace
serial.delete Remove serial port object from memory
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>serial.fgetl</code></td>
<td>Read line of text from device and discard terminator</td>
</tr>
<tr>
<td><code>serial fgets</code></td>
<td>Read line of text from device and include terminator</td>
</tr>
<tr>
<td><code>serial fopen</code></td>
<td>Connect serial port object to device</td>
</tr>
<tr>
<td><code>serial fprintf</code></td>
<td>Write text to device</td>
</tr>
<tr>
<td><code>serial fread</code></td>
<td>Read binary data from device</td>
</tr>
<tr>
<td><code>serial fscanf</code></td>
<td>Read data from device, and format as text</td>
</tr>
<tr>
<td><code>serial fwrite</code></td>
<td>Write binary data to device</td>
</tr>
<tr>
<td><code>serial get</code></td>
<td>Serial port object properties</td>
</tr>
<tr>
<td><code>serial isvalid</code></td>
<td>Determine whether serial port objects are valid</td>
</tr>
<tr>
<td><code>serial length</code></td>
<td>Length of serial port object array</td>
</tr>
<tr>
<td><code>serial load</code></td>
<td>Load serial port objects and variables into MATLAB workspace</td>
</tr>
<tr>
<td><code>serial save</code></td>
<td>Save serial port objects and variables to MAT-file</td>
</tr>
<tr>
<td><code>serial set</code></td>
<td>Configure or display serial port object properties</td>
</tr>
<tr>
<td><code>serial size</code></td>
<td>Size of serial port object array</td>
</tr>
<tr>
<td><code>serial break</code></td>
<td>Send break to device connected to serial port</td>
</tr>
<tr>
<td><code>stopasync</code></td>
<td>Stop asynchronous read and write operations</td>
</tr>
</tbody>
</table>
Alphabetical List

Arithmetic Operators: + - * / \ ^ ’
Relational Operators: < > <= >= == ~=
Logical Operators: Elementwise & | ~
Logical Operators: Short-circuit && ||
Special Characters: [ ] ( ) { } = ’ . ... ; : % ! @
colon (:)
abs
accumarray
acos
acosd
acosh
acot
acotd
acoth
acsc
acscd
acsch
actxcontrol
actxcontrollist
actxcontrolselect
actxGetRunningServer
actxserver
addCause (MException)
addevent
addframe
adddispatcher (handle)
addOptional (inputParser)
addParamValue (inputParser)
addpath
addpref
addprop (dynamicprops)
addproperty
addRequired (inputParser)
addsample
addsampletocollection
addtodate
addts
airy
align
align
alim
all
allchild
alpha
alphamap
amd
ancestor
and
angle
annotation
Annotation Arrow Properties
Annotation Doublearrow Properties
Annotation Ellipse Properties
Annotation Line Properties
Annotation Rectangle Properties
Annotation Textarrow Properties
Annotation Textbox Properties
ans
any
area
Areaseries Properties
arrayfun
ascii
asec
asecd
asech
asin
asind
asinh
assert
assignin
atan
atan2
atand
atanh
audiodevinfo
audioplayer
audiorecorder
aufinfo
auread
auwrite
avifile
aviinfo
aviread
axes
Axes Properties
axis
balance
bar, barh
bar3, bar3h
Barseries Properties
baryToCart
base2dec
beep
bench
besselh
besseli
besselj
besselk
bessely
beta
betainc
betaincinv
betaln
bicg
bicgstab
bicgstabl
bin2dec
binary
bitand
bitcmp
bitget
bitmax
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
brush
bsxfun
builddocsearchdb
builtin
bvp4c
bvp5c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit
cholinc
cholupdate
circshift
circumcenters
cla
clabel
class
classdef
clc
clear
clearvars
serial.clear
clf
clipboard
clock
close
close (avifile)
close (ftp)
closereq
cmopts
cmpermute
cmunique
colamd
colorbar
colordef
colormap
colormapeditor
ColorSpec (Color Specification)
colperm
comet
comet3
commandhistory
commandwindow
compan
compass
complex
computer
cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contourgroup Properties
contourslice
contrast
conv
conv2
convhull
convhulln
convn
copyfile
copyobj
correcoef
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
cputime
create (RandStream)
createClassFromWsdl
createCopy (inputParser)
createSoapMessage
cross
csc
cscd
csch
csvread
csvwrite
ctranspose (timeseries)
cumprod
cumsum
cumtrapz
curl
customverctrl
cylinder
daqread
daspect
datacursormode
datatipinfo
date
datenum
datestr
datetick
datevec
dbclear
dbcont
dbdown
dblquad
dbmex
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeget
ddesd
ddeset
deal
debblank
debug
dec2base
dec2bin
dec2hex
decic
deconv
del2
DelaunayTri
DelaunayTri
delaunay
delaunay3
delaunayn
delete
delete (COM)
delete (ftp)
delete (handle)
serial.delete
delete (timer)
deleteproperty
delevent
delsample
delsamplefromcollection
demo
depdir
depfun
det
detrend
detrend (timeseries)
deval
diag
dialog
diary
diff
diffuse
dir
dir (ftp)
disp
disp (memmapfile)
disp (MException)
serial.disp
disp (timer)
display
dither
divergence
dlmread
dlmwrite
dmperm
doc
docopt
docsearch
dos
dot
double
dragrect
drawnow
dsearch
dsearchn
dynamicprops
echo
echodemo
edgeAttachments
edges
edit
eig
eigs
ellipj
ellipke
ellipsoid
else
elseif
enableservice
end
eomday
eps
eq
eq (MException)
erf, erfc, erfcx, erfinv, erfcinv
error
errorbar
Errorbarseries Properties
errordlg
etime
etree
etreeplot
eval
evalc
evalin
event.EventData
event.PropertyEvent
event.listener
event.proplistener
eventlisteners
events
events (COM)
Execute
exifread
exist
exit
exp
expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh
ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
faceNormals
factor
factorial
false
fclose
serial.fclose
feather
featureEdges
feof
ferror
feval
Feval (COM)
fft
fft2
fftn
fftshift
fftw
fgetl
serial.fgetl
fgets
serial.fgets
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
File Formats
filemarker
fileparts
fileread
filesep
fill
fill3
filter
filter (timeseries)
filter2
find
findall
findfigs
findobj
findobj (handle)
findprop (handle)
findstr
finish
fitsinfo
fitsread
fix
flipdim
fliplr
flipud
floor
flow
fminbnd
fminsearch
fopen
serial.fopen
for
format
fplot
fprintf
serialfprintf
frame2im
fread
serial.fread
freeBoundary
freqspace
frewind
fscanf
serial.fscanf
fseek
ftell
ftp
cfull
fullfile
func2str
function
function_handle (@)
functions
funm
fwrite
serial.fwrite
fzero
gallery
gamma, gammainc, gammainv
gammaincinv
gca
gcbf
gcbo
gcd
gcf
gco
ge
ge
genpath
genvarname
get
get (COM)
get (hgsetget)
get (memmapfile)
get (RandStream)
serial.get
get (timer)
get (timeseries)
get (tscollection)
getabstime (timeseries)
getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getDefautStream (RandStream)
getdisp (hgsetget)
getenv
getField
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getReport (MException)
getsampleusingtime (timeseries)
getsampleusingtime (tscollection)
gettimeseriesnames
gettafteratevent
gettafterevent
gettaevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents
GetVariable
GetWorkspaceData
ginput
global
gmres
gplot
grcode
gradient
graymon
grid
griddata
griddata3
griddatan
gsvd
gt
gtext
guidata
guide
guihandles
gunzip
gzip
hadamard
handle
hankel
hdf
hdf5
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
hex2dec
hex2num
hgexport
hgggroup
Hggroup Properties
hglode
hgsave
hgsetget
hgtransform
Hgtransform Properties
hidden
hilb
hist
histc
hold
home
horzcat
horzcat (tscollection)
hostid
hsv2rgb
hypot
i
idealfilter (timeseries)
idivide
if
ifft
ifft2
ifftn
ifftshift
ilu
im2frame
im2java
imag
image
Image Properties
imagesc
imapprox
imfinfo
imformats
import
importdata
imread
imwrite
incenters
inOutStatus
ind2rgb
ind2sub
Inf
inferiorto
info
inline
inmem
inpolygon
input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8, int16, int32, int64
interfaces
interp1
interp1q
interp2
interp3
interpft
interpn
interpstreamspeed
intersect
intmax
intmin
intwarning
inv
invhilb
invoke
ipermute
iqr (timeseries)
is*
is
isa
isappdata
iscell
iscellstr
ischar
iscom
isdir
isEdge
isempty
isempty (timeseries)
isempty (tscollection)
isequal
isequal (MException)
isequalwithethequalnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishold
isinf
isinteger
isinterface
isjava
isKey (Map)
iskeyword
isletter
islogical
ismac
ismember
ismethod
isnan
isnumeric
isobject
isocaps
isocolors
isonormals
isosurface
ispc
ispref
isprime
isprop
isreal
isscalar
issorted
isseps
issepsp
isstr
isstrprop
isstruct
isstudent
isunix
isvalid (handle)
serial.isvalid
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
javaMethod
javaMethodEDT
javaObject
javaObjectEDT
javarmpath
keyboard
keys (Map)
kron
last (MException)
lasterr
lasterror
lastwarn
lcm
ldl
ldivide, rdivide
le
legend
legendre
length
length (Map)
serial.length
length (timeseries)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec (Line Specification)
linkaxes
linkdata
linkprop
linsolve
linspace
list (RandStream)
listdlg
listfonts
load
load (COM)
serial.load
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor
lower
ls
lscov
lsqlin
lsqnonneg
lsqr
lt
lu
luinc
magic
makehgtform
containers.Map
mat2cell
mat2str
material
matlabcolon (matlab:)
matlabrc
matlabroot
matlab (UNIX)
matlab (Windows)
max
max (timeseries)
MaximizeCommandWindow
maxNumCompThreads
mean
mean (timeseries)
median
median (timeseries)
memmapfile
memory
menu
mesh, meshc, meshz
meshgrid
meta.class
meta.class.fromName
meta.DynamicProperty
meta.event
meta.method
meta.package
meta.package.fromName
meta.package.getAllPackages
meta.property
metaclass
methods
methodview
mex
mex.getCompilerConfigurations
MException
mexext
mfilename
mget
min
min (timeseries)
MinimizeCommandWindow
minres
mislocked
mkdir
mkdir (ftp)
mkpp
mldivide \, mrdivide /
mlint
mlintrpt
mlock
mmfileinfo
mmreader
mmreader.isPlatformSupported
mod
mode
more
move
movefile
movegui
movie
movie2avi
mput
msgbox
mtimes
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin, nargout
nargoutchk
native2unicode
nchoosek
ndgrid
ndims
ne
nearestNeighbor
ne (MException)
neighbors
NET
NET.addAssembly
NET.convertArray
NET.createArray
NET.createGeneric
NET.GenericClass
NET.GenericClass
netcdf
netcdf.abort
netcdf.close
netcdf.copyAtt
netcdf.create
netcdf.defDim
netcdf.defVar
netcdf.delAtt
netcdf.endDef
netcdf.getAtt
netcdf.getConstant
netcdf.getConstantNames
netcdf.getVar
netcdf.inq
netcdf.inqAtt
netcdf.inqAttID
netcdf.inqAttName
netcdf.inqDim
netcdf.inqDimID
netcdf.inqLibVers
netcdf.inqVar
netcdf.inqVarID
netcdf.open
netcdf.putAtt
netcdf.putVar
netcdf.reDef
netcdf.renameAtt
netcdf.renameDim
netcdf.renameVar
netcdfsetDefaultFormat
netcdf.setFill
netcdf.sync
newplot
nextpow2
nnz
noanimate
nonzeros
norm
normest
not
notebook
notify (handle)
now
nthroot
null
num2cell
num2hex
num2str
numel
nzmax
ode15i
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
odefile
odeget
odeset
odextend
onCleanup
ones
open
openfig
opengl
openvar
optimget
optimset
or
ordeig
orderfields
ordqz
ordschur
orient
orth
otherwise
pack
padecoef
pagesetupdlg
pan
pareto
parfor
parse (inputParser)
parseSoapResponse
partialpath
pascal
patch
Patch Properties
path
path2rc
pathsep
pathtool
pause
pbaspect
pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie
pie3
pinv
planerot
playshow
plot
plot (timeseries)
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
pointLocation
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval
polyvalm
pow2
power
ppval
prefdir
preferences
primes
print, printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
properties
propertyeditor
psi
publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quad2d
quadgk
quadl
quadv
questdlg
quit
Quit (COM)
quiver
quiver3
Quivergroup Properties
qz
rand
rand (RandStream)
randi
randi (RandStream)
randn
randn (RandStream)
randperm
randperm (RandStream)
RandStream
RandStream (RandStream)
rank
rat, rats
rbbox
rcond
read
readasync
real
reallog
realmax
realmin
realpow
realsqrt
record
rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp, regexpi
regexprep
regextranslate
registerevent
rehash
release
relationaloperators (handle)
rem
remove (Map)
removets
rename
repmat
resample (timeseries)
resample (tscollection)
reset
reset (RandStream)
reshape
residue
restoredefaultpath
rethrow
rethrow (MException)
return
rgb2hsv
rgb2ind
rgbplot
ribbon
rmappdata
rmdir
rmdir (ftp)
rmfield
rmpath
rmpref
root object
Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rref
rsf2csf
run
save
save (COM)
serial.save
saveas
saveobj
savepath
scatter
scatter3
Scattergroup Properties
schur
script
sec
secd
sech
selectmoveresize
semilogx, semilogy
sendmail
serial
serialbreak
set
set (COM)
set (hgsetget)
set (RandStream)
serial.set
set (timer)
set (timeseries)
set (tscollection)
setabstime (timeseries)
setabstime (tscollection)
setappdata
setDefaultStream (RandStream)
setdiff
setdisp (hgsetget)
setenv
setfield
setinterpmethod
setpixmapposition
setpref
setstr
settimesteriesnames
setxor
shading
shg
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size (Map)
serial.size
size (timeseries)
size
size (tscollection)
slice
smooth3
snapnow
sort
sortrows
sound
soundsc
spalloc
sparse
spaugment
spconvert
spdiags
specular
speye
spfun
sph2cart
sphere
spinmap
spline
spones
spparms
sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stairs
Stairseries Properties
start
startat
startup
std
std (timeseries)
stem
stem3
Stemseries Properties
stop
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp, strcmpi
stream2
stream3
streamline
streamparticles
streamribbon
streamslice
streamtube
strfind
strings
strjust
strmatch
strncmp, strncmpi
strread
strrep
strtok
strtrim
struct
struct2cell
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substruct
subvolume
sum
sum (timeseries)
superiorto
support
surf, surfc
surf2patch
surface
Surface Properties
Surfaceplot Properties
surfl
surfnorm
svd
svds
swapbytes
switch
symamd
symbsfact
symmlq
symrcm
symvar
synchronize
syntax
system
tan
tand
tanh
tar
tempdir
tempname
teramesh
texlabel
text
Text Properties
textread
textscan
textrap
tfqmr
throw (MException)
throwAsCaller (MException)
tic, toc
timer
timerfind
timerfindall
timeseries
title
todatenum
toeplitz
toolboxdir
trace
transpose (timeseries)
trapz
treelayout
treeplot
tril
trimesh
triplequad
triplot
TriRep
TriRep
TriScatteredInterp
TriScatteredInterp
trisurf
triu
true
try
tscollection
tsdata.event
tsearch
tsearchn
tsprps
tstool
type
typecast
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
UiControl Properties
uigetdir
uigetfile
uigetpref
uimport
uimenu
Uimenu Properties
uint8, uint16, uint32, uint64
uiopen
uipanel
Uipanel Properties
uipushtool
Uipushtool Properties
uiputfile
uiresume
uisave
uisetcolor
uisetfont
uisetpref
uistack
uitable
Uitable Properties
uitoggletool
Uitoggletool Properties
uitoolbar
Uitoolbar Properties
uiwait
undocheckout
unicode2native
union
unique
unix
unloadlibrary
unmesh
unmkpp
unregisterallevents
unregisterevent
untar
unwrap
unzip
upper
urlread
urlwrite
usejava
userpath
validateattributes
validatestring
values (Map)
vander
var
var (timeseries)
varargin
varargout
vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (timeseries)
vertcat (tscollection)
vertexAttachments
view
viewmtx
visdiff
volumebounds
voronoi
voronoiDiagram
voronoin
wait
waitbar
waitfor
waitforbuttonpress
warndlg
warning
waterfall
wavinfo
wavplay
wavread
wavrecord
wavwrite
weekday
what
whatsnew
which
while
whitebg
who, whos
wilkinson
winopen
winqueryreg
wk1info
wk1read
wk1write
workspace
xlabel, ylabel, zlabel
xlim, ylim, zlim
xlsinfo
xlsread
xlswrite
xmlread
xmlwrite
xor
xslt
zeros
zip
zoom
Purpose
Consolidate workspace memory

Syntax
pack
pack filename
pack('filename')

Description
pack frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from your base and global workspaces are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, []). The MATLAB software temporarily stores your workspace data in a file called tp####.mat (where #### is a numeric value) that is located in your temporary directory. (You can use the command dir(tempdir) to see the files in this directory).

pack filename frees space in memory, temporarily storing workspace data in a file specified by filename. This file resides in your current working directory and, unless specified otherwise, has a .mat file extension.

pack('filename') is the function form of pack.

Remarks
You can only run pack from the MATLAB command line.
If you specify a filename argument, that file must reside in a directory for which you have write permission.

The pack function does not affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory. Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the Out of memory message from MATLAB, the pack function may find you some free memory without forcing you to delete variables. The pack function frees space by
• Saving all variables in the base and global workspaces to a temporary file.
• Clearing all variables and functions from memory.
• Reloading the base and global workspace variables back from the temporary file and then deleting the file.

If you use `pack` and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, you can allocate larger matrices earlier in the MATLAB session and use these system-specific tips:

• When running MATLAB on The Open Group UNIX platforms, ask your system manager to increase your swap space.
• On Microsoft Windows platforms, increase virtual memory using the Windows Control Panel.

To maintain persistent variables when you run `pack`, use `mlock` in the function.

**Examples**

Change the current directory to one that is writable, run `pack`, and return to the previous directory.

```matlab
cwd = pwd;
cd(tempdir);
pack
cd(cwd)
```

**See Also**
clear, memory
Purpose
Padé approximation of time delays

Syntax
[num,den] = padecoef(T,N)

Description
[num,den] = padecoef(T,N) returns the Nth-order Padé approximation of the continuous-time delay T in transfer function form. The row vectors num and den contain the numerator and denominator coefficients in descending powers of T. Both are Nth-order polynomials.

Class support for input T:
float: double, single

Class Support
Input T support floating-point values of type single or double.

References

See Also
pade
Purpose

Page setup dialog box

Syntax

dlg = pagesetupdlg(fig)

**Note**  This function is obsolete. Use printpreview instead.

Description

dlg = pagesetupdlg(fig) creates a dialog box from which a set of pagelayout properties for the figure window, fig, can be set.

pagesetupdlg implements the "Page Setup..." option in the **Figure** File Menu.

pagesetupdlg supports setting the layout for a single figure. fig must be a single figure handle, not a vector of figures or a simulink diagram.
See Also

printdlg, printpreview, printopt
Purpose
Pan view of graph interactively

GUI Alternatives
Use the Pan tool on the figure toolbar to enable and disable pan mode on a plot, or select Pan from the figure’s Tools menu. For details, see “Panning — Shifting Your View of the Graph” in the MATLAB Graphics documentation.

Syntax

```matlab
pan on
pan xon
pan yon
pan off
pan
pan(figure_handle,...)
```

```matlab
h = pan(figure_handle)
```

Description

- `pan on` turns on mouse-based panning in the current figure.
- `pan xon` turns on panning only in the x direction in the current figure.
- `pan yon` turns on panning only in the y direction in the current figure.
- `pan off` turns panning off in the current figure.
- `pan` toggles the pan state in the current figure on or off.
- `pan(figure_handle,...)` sets the pan state in the specified figure.
- `h = pan(figure_handle)` returns the figure’s pan mode object for the figure `figure_handle` for you to customize the mode’s behavior.

Using Pan Mode Objects

Access the following properties of pan mode objects via `get` and modify some of them using `set`:

- `Enable 'on' | 'off'` — Specifies whether this figure mode is currently enabled on the figure
- `Motion 'horizontal' | 'vertical' | 'both'` — The type of panning enabled for the figure
• **FigureHandle <handle>** — The associated figure handle, a read-only property that cannot be set

**Pan Mode Callbacks**

You can program the following callbacks for pan mode operations.

• **ButtonDownFilter <function_handle>** — Function to intercept ButtonDown events

  The application can inhibit the panning operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```matlab
  function [res] = myfunction(obj,event_obj)
  % obj handle to the object that has been clicked on
  % event_obj event data (empty in this release)
  % res [output] a logical flag to determine whether the pan
  % operation should take place or the 'ButtonDownFcn'
  % property of the object should take precedence
  ```

• **ActionPreCallback <function_handle>** — Function to execute before panning

  Set this callback to if you need to execute code when a pan operation begins. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```matlab
  function myfunction(obj,event_obj)
  % obj handle to the figure that has been clicked on
  % event_obj object containing struct of event data
  ```

  The event data struct has the following field:

  | Axes          | The handle of the axes that is being panned |
• **ActionPostCallback** `<function_handle>` — Function to execute after panning

Set this callback if you need to execute code when a pan operation ends. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

```matlab
function myfunction(obj, event_obj)
    % obj          handle to the figure that has been clicked on
    % event_obj    object containing struct of event data (same as the
                   % event data of the 'ActionPreCallback' callback)
```

### Pan Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- **flags = isAllowAxesPan(h, axes)** — Function querying permission to pan axes

  Calling the function `isAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a pan operation is permitted on the axes objects.

- **setAllowAxesPan(h, axes, flag)** — Function to set permission to pan axes

  Calling the function `setAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a pan operation on the axes objects.

- **info = getAxesPanMotion(h, axes)** — Function to get style of pan operations

  Calling the function `getAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, as input will return a character cell array of the same dimension as the axes handle vector, which indicates the type of pan operation for each axes. Possible values for the type of operation are 'horizontal', 'vertical' or 'both'.

2-2685
setAxesPanMotion(h, axes, style) — Function to set style of pan operations

Calling the function setAxesPanMotion on the pan object, h, with a vector of axes handles, axes, and a character array, style, sets the style of panning on each axes.

Examples

Example 1 — Entering Pan Mode

Plot a graph and turn on Pan mode:

```
plot(magic(10));
pan on
% pan on the plot
```

Example 2 — Constrained Pan

Constrain pan to x-axis using set:

```
plot(magic(10));
h = pan;
set(h,'Motion','horizontal','Enable','on');
% pan on the plot in the horizontal direction.
```

Example 3 — Constrained Pan in Subplots

Create four axes as subplots and give each one a different panning behavior:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = pan;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesPan(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesPanMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
```
contour(peaks);
setAxesPanMotion(h,ax4,'vertical');
% pan on the plots.

Example 4 — Coding a ButtonDown Callback

Create aButtonDown callback for pan mode objects to trigger. Copy the following code to a new M-file, execute it, and observe panning behavior:

```matlab
function demo
    % Allow a line to have its own 'ButtonDownFcn' callback.
    hLine = plot(rand(1,10));
    set(hLine,'ButtonDownFcn','disp(''This executes'')');
    set(hLine,'Tag','DoNotIgnore');
    h = pan;
    set(h,'ButtonDownFilter',@mycallback);
    set(h,'Enable','on');
    % mouse click on the line
    %
    function [flag] = mycallback(obj,event_obj)
    % If the tag of the object is 'DoNotIgnore', then return true.
    % Indicate what the target is
    disp([''Clicked ' get(obj,'Type') ' object''])
    objTag = get(obj,'Tag');
    if strcmpi(objTag,'DoNotIgnore')
        flag = true;
    else
        flag = false;
    end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-ButtonDown events for pan mode objects to trigger. Copy the following code to a new M-file, execute it, and observe panning behavior:

```matlab
function demo
    % Listen to pan events
```
plot(1:10);
h = pan;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');

function myprecallback(obj,evd)
disp('A pan is about to occur.');
end

function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%2.2f %2.2f].',newLim));
end

Example 6 — Creating a Context Menu for Pan Mode

Coding a context menu that lets the user to switch to Zoom mode by
right-clicking:

figure; plot(magic(10));
hCM = uicontextmenu;
hMenu = uimenu('Parent',hCM,'Label','Switch to zoom', ...
    'Callback','zoom(gcf,''on'')');
hPan = pan(gcf);
set(hPan,'UIContextMenu',hCM);
pan('on')

You cannot add items to the built-in pan context menu, but you can replace it with your own.

Remarks

You can create a pan mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.
**Note** Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different pan behaviors to different subplot axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

**See Also**

`zoom`, `linkaxes`, `rotate3d`

“Object Manipulation” on page 1-107 for related functions
Purpose

Pareto chart

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```matlab
pareto(Y)
p pareto(Y,names)
p pareto(Y,X)
H = pareto(...)
```

Description

Pareto charts display the values in the vector Y as bars drawn in descending order. Values in Y must be nonnegative and not include NaNs. Only the first 95% of the cumulative distribution is displayed.

`pareto(Y)` labels each bar with its element index in Y and also plots a line displaying the cumulative sum of Y.

`pareto(Y,names)` labels each bar with the associated name in the string matrix or cell array names.

`pareto(Y,X)` labels each bar with the associated value from X.

`pareto(ax,..)` plots a Pareto chart in existing axes ax rather than GCA.

`H = pareto(...)` returns a combination of patch and line object handles.

Examples

Example 1:

Examine the cumulative productivity of a group of programmers to see how normal its distribution is:
```matlab
codelines = [200 120 555 608 1024 101 57 687];
coders = ...
{'Fred','Ginger','Norman','Max','Julia','Wally','Heidi','Pat'};
pareto(codelines, coders)
title('Lines of Code by Programmer')
```

Example 2:

Generate a vector, $X$, representing diagnostic codes with values from 1 to 10 indicating various faults on devices emerging from a production line:

```matlab
X = min(round(abs(randn(100,1)*4))+1,10);
```

Plot a Pareto chart showing the frequency of failure for each diagnostic code from the most to the least common:

```matlab
pareto(hist(X))
```
**Remarks**

You can use `pareto` to display the output of `hist`, even for vectors that include negative numbers. Because only the first 95 percent of values are displayed, one or more of the smallest bars may not appear. If you extend the `Xlim` of your chart, you can display all the values, but the new bars will not be labeled.

You cannot place datatips (use the Datacursor tool) on graphs created with `pareto`.

**See Also**

`hist`, `bar`
Purpose

Parallel for-loop

Syntax

parfor loopvar = initval:endval; statements; end
parfor (loopvar = initval:endval, M); statements; end

Description

parfor loopvar = initval:endval; statements; end executes a series of MATLAB commands denoted here as statements for values of loopvar between initval and endval, inclusive, which specify a vector of increasing integer values. Unlike a traditional for-loop, there is no guarantee of the order in which the loop iterations are executed.

The general format of a parfor statement is:

```
parfor loopvar = initval:endval
        <statements>
end
```

Certain restrictions apply to the statements to ensure that the iterations are independent, so that they can execute in parallel. If you have the Parallel Computing Toolbox™ software, the iterations of statements can execute in parallel on separate MATLAB workers on your multi-core computer or computer cluster.

To execute the loop body in parallel, you must open a pool of MATLAB workers using the `matlabpool` function, which is available in Parallel Computing Toolbox.

```
parfor (loopvar = initval:endval, M); statements; end
```

executes statements in a loop using a maximum of M MATLAB workers to evaluate statements in the body of the parfor-loop. Input variable M must be a nonnegative integer. By default, MATLAB uses up to as many workers as it finds available.

When any of the following are true, MATLAB does not execute the loop in parallel:

- There are no workers in a MATLAB pool
- You set M to zero
• You do not have Parallel Computing Toolbox

If you have Parallel Computing Toolbox, you can read more about parfor and matlabpool by typing

    doc distcomp/parfor
    doc distcomp/matlabpool

**Examples**  
Perform three large eigenvalue computations using three computers or cores:

    matlabpool(3)
    parfor i=1:3, c(:,i) = eig(rand(1000)); end

**See Also**  
for
**Purpose**

Parse and validate named inputs

**Syntax**

```matlab
p.parse(arglist)
parse(p, arglist)
```

**Description**

`p.parse(arglist)` parses and validates the inputs named in `arglist`. `parse(p, arglist)` is functionally the same as the syntax above.

For more information on the `inputParser` class, see “Parsing Inputs with `inputParser`” in the MATLAB Programming Fundamentals documentation.

**Examples**

Write an M-file function called `publish_ip`, based on the MATLAB `publish` function, to illustrate the use of the `inputParser` class. Construct an instance of `inputParser` and assign it to variable `p`:

```matlab
function publish_ip(script, varargin)
p = inputParser; % Create an instance of the inputParser class.
```

Add arguments to the schema. See the reference pages for the `addRequired`, `addOptional`, and `addParamValue` methods for help with this:

```matlab
    p.addRequired('script', @ischar);
p.addOptional('format', 'html', ...
        @(x)any(strcmpi(x,{'html','ppt','xml','latex'}))));
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Call the `parse` method of the object to read and validate each argument in the schema:

```matlab
    p.parse(script, varargin{:});
```

Execution of the `parse` method validates each argument and also builds a structure from the input arguments. The name of the structure is
Results, which is accessible as a property of the object. To get the value of any input argument, type

\[ p.\text{Results}.\text{argname} \]

Continuing with the publish_ip exercise, add the following lines to your M-file:

```matlab
% Parse and validate all input arguments.
p.parse(script, varargin{:});

% Display the value for maxHeight.
disp(sprintf('\nThe maximum height is %d\n', p.Results.maxHeight))

% Display all arguments.
disp 'List of all arguments:'
disp(p.Results)
```

When you call the program, MATLAB assigns those values you pass in the argument list to the appropriate fields of the Results structure. Save the M-file and execute it at the MATLAB command prompt with this command:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', 'C:/matlab/test', ...
' maxWidth', 500, 'maxHeight', 300);
```

The maximum height is 300.

List of all arguments:

- format: 'ppt'
- maxHeight: 300
- maxWidth: 500
- outputDir: 'C:/matlab/test'
- script: 'ipscript.m'

**See Also**

inputParser, addRequired(inputParser),
addOptional(inputParser), addParamValue(inputParser),
createCopy(inputParser)
**Purpose**
Convert response string from SOAP server into MATLAB types

**Syntax**
```matlab
parseSoapResponse(response)
```

**Description**
`parseSoapResponse(response)` extracts data from `response` a string returned by a SOAP server from the `callSoapService` function, and converts it to appropriate MATLAB classes (types).

**Examples**
This example uses `parseSoapResponse` in conjunction with other SOAP functions to retrieve information about books from a library database, specifically, the author’s name for a given book title.

**Note** The example does not use an actual endpoint; therefore, you cannot run it. The example only illustrates how to use the SOAP functions.

```matlab
% Create the message:
message = createSoapMessage(...
    'urn:LibraryCatalog',...
    'getAuthor',...
    {'In the Fall'},...
    {'nameToLookUp'},...
    {'{http://www.w3.org/2001/XMLSchema}string'},...
    'rpc');
%
% Send the message to the service and get the response:
response = callSoapService(...
    'http://test/soap/services/LibraryCatalog',...
    'urn:LibraryCatalog#getAuthor',...
    message)
%
% Extract MATLAB data from the response
author = parseSoapResponse(response)
```

MATLAB returns:
author = Kate Alvin

where author is a char class (type).

See Also
callSoapService, createClassFromWsdl, createSoapMessage,
urlread, xmlread

“Using Web Services with MATLAB” in the MATLAB External
Interfaces documentation
Purpose

Partial pathname description

Description

A partial pathname is the last portion of a full pathname for a file in a directory on the MATLAB search path. Use a partial pathname to locate private and method files, which are usually hidden. You can also use a partial pathname to restrict the search for files to a portion of the path when a file is overloaded, that is, when more than one file with the given name exists in different directories.

A partial pathname consists of the last directory or last several directories of the full pathname, with each directory separated by /, and followed by the filename. For example, matfun/trace, private/cancel1, and demos/clown.mat are valid partial pathnames. Specifying the @ in method directory names is optional.

Partial pathnames also make it easy to specify a toolbox or a file in a directory on your path, independent of the location where MATLAB is installed.

Many commands accept partial pathnames instead of a full pathname. Some of these commands are: dbclear, dbtype, dbstop, edit, exist, fopen, help, load, type, what, which. The reference page for a function typically specifies if the function accepts partial pathnames.

Examples

The following example uses a partial pathname:

```
what graph2d/@figobj
```

```
M-files in directory
matlabroot\toolbox\matlab\graph2d\@figobj

deselectall  enddrag  middrag  subsref
doclick      figobj    set
doresize      get       subsasgn
```

The @ in the class directory name @figobj is not necessary. You get the same response from the following command:

```
what graph2d/figobj
```
See Also

fileparts, matlabroot, path

Purpose

Pascal matrix

Syntax

A = pascal(n)
A = pascal(n,1)
A = pascal(n,2)

Description

A = pascal(n) returns the Pascal matrix of order n: a symmetric positive definite matrix with integer entries taken from Pascal’s triangle. The inverse of A has integer entries.

A = pascal(n,1) returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. It is involutary, that is, it is its own inverse.

A = pascal(n,2) returns a transposed and permuted version of pascal(n,1). A is a cube root of the identity matrix.

Examples

pascal(4) returns

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 20 \\
\end{array}
\]

A = pascal(3,2) produces

\[
A = \\
\begin{array}{ccc}
1 & 1 & 1 \\
-2 & -1 & 0 \\
1 & 0 & 0 \\
\end{array}
\]

See Also

chol
**Purpose**
Create patch graphics object

**Syntax**
- `patch(X,Y,C)`
- `patch(X,Y,Z,C)`
- `patch(FV)`
- `patch(...'PropertyName',propertyvalue...)`
- `patch('PropertyName',propertyvalue,...)`
- `handle = patch(...)`

**Description**
`patch` is the low-level graphics function for creating patch graphics objects. A patch object is one or more polygons defined by the coordinates of its vertices. You can specify the coloring and lighting of the patch. See “Creating 3-D Models with Patches” in *MATLAB 3-D Visualization* for more information on using patch objects.

`patch(X,Y,C)` adds the filled two-dimensional patch to the current axes. The elements of X and Y specify the vertices of a polygon. If X and Y are matrices, MATLAB draws one polygon per column. C determines the color of the patch. It can be a single ColorSpec, one color per face, or one color per vertex (see “Specifying Patch Properties” on page 2-2703 and “Color Data Interpretation” on page 2-2704). If C is a 1-by-3 vector, it is assumed to be an RGB triplet, specifying a color directly.

`patch(X,Y,Z,C)` creates a patch in three-dimensional coordinates.

`patch(FV)` creates a patch using structure FV, which contains the fields vertices, faces, and optionally facevertexcdata. These fields correspond to the Vertices, Faces, and FaceVertexCData patch properties.

`patch(...'PropertyName',propertyvalue...)` follows the X, Y, (Z), and C arguments with property name/property value pairs to specify additional patch properties.

`patch('PropertyName',propertyvalue,...)` specifies all properties using property name/property value pairs. This form enables you to omit the color specification because MATLAB uses the default face color and edge color unless you explicitly assign a value to the FaceColor and EdgeColor properties. This form also allows you to specify the
patch using the Faces and Vertices properties instead of x-, y-, and z-coordinates. See “Examples” on page 2-2706 for more information.

```matlab
handle = patch(...)```

returns the handle of the patch object it creates.

**Remarks**

Unlike high-level area creation functions, such as `fill` or `area`, `patch` does not check the settings of the figure and axes `NextPlot` properties. It simply adds the patch object to the current axes.

If the coordinate data does not define closed polygons, `patch` closes the polygons. The data can define concave or intersecting polygons. However, if the edges of an individual patch face intersect themselves, the resulting face may or may not be completely filled. In that case, it is better to divide the face into smaller polygons.

**Specifying Patch Properties**

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

There are two patch properties that specify color:

- **CData** — Use when specifying x-, y-, and z-coordinates (`XData`, `YData`, and `ZData`).
- **FaceVertexCData** — Use when specifying vertices and connection matrix (`Vertices` and `Faces`).

The `CData` and `FaceVertexCData` properties accept color data as indexed or true color (RGB) values. See the `CData` and `FaceVertexCData` property descriptions for information on how to specify color.

Indexed color data can represent either direct indices into the colormap or scaled values that map the data linearly to the entire colormap (see the `caxis` function for more information on this scaling). The `CDataMapping` property determines how MATLAB interprets indexed color data.
**Color Data Interpretation**

You can specify patch colors as:

- A single color for all faces
- One color for each face, enabling flat coloring
- One color for each vertex, enabling interpolated coloring

The following tables summarize how MATLAB interprets color data defined by the `CData` and `FaceVertexCData` properties, respectively.

In the following table, \( m \) (the number of rows) represents the number of vertices per face and \( n \) (the number of columns) represents the total number of faces (of that type of shape) per patch object. For example, if you want to create a patch object with four triangular faces, the `XData`, `YData`, and `ZData` would be 3-by-4 matrices. See “Creating Pictures with Multiple Patch Objects” on page 2-2709 for more information.
### Interpretation of the CData Property

<table>
<thead>
<tr>
<th>[X,Y,Z]Data Dimensions</th>
<th>CData Required for</th>
<th>Results Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-by-n</td>
<td>Indexed</td>
<td>True Color</td>
</tr>
<tr>
<td>scalar</td>
<td>1-by-1-by-3</td>
<td>Use the single color specified for all patch faces. Edges can be only a single color.</td>
</tr>
<tr>
<td>1-by-n</td>
<td>1-by-n-by-3</td>
<td>Use one color for each patch face. Edges can be only a single color.</td>
</tr>
<tr>
<td>m-by-n (n &gt;= 4)</td>
<td>m-by-n-by-3</td>
<td>Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated.</td>
</tr>
</tbody>
</table>

In the following table, \( m \) represents the number of faces in a patch object, \( n \) represents the number of vertices per face, and \( k \) represents the total number of unique vertices per object. In addition, \( 3 \leq n \leq k \). See “Creating Pictures with Multiple Patch Objects” on page 2-2709 for more information.
Interpretation of the FaceVertexCData Property

<table>
<thead>
<tr>
<th>Faces</th>
<th>Vertices</th>
<th>FaceVertexCData Required for</th>
<th>Results Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions</td>
<td>Dimensions</td>
<td>Indexed</td>
<td>True Color</td>
</tr>
<tr>
<td>m-by-n</td>
<td>k-by-3</td>
<td>scalar</td>
<td>1-by-3</td>
</tr>
<tr>
<td>m-by-n</td>
<td>k-by-3</td>
<td>k-by-1</td>
<td>k-by-3</td>
</tr>
<tr>
<td>m-by-n</td>
<td>k-by-3</td>
<td>m-by-1</td>
<td>m-by-3</td>
</tr>
</tbody>
</table>

Examples

The next two examples create a patch object using two different methods:

- Specifying x-, y-, and z-coordinates and color data (XData, YData, ZData, and CData properties)
- Specifying vertices, the connection matrix, and color data (Vertices, Faces, FaceVertexCData, and FaceColor properties)

Specifying X-, Y-, and Z-Coordinates

The first approach specifies the coordinates of each vertex. In this example, the coordinate data, a 3-by-2 matrix, defines two triangular faces, each having three vertices. Using true color, the top face is set to white and the bottom face to gray.

```matlab
x = [0 0;
     0 1;
     1 1];
```
y = [1 1; 2 2; 2 1];
z = [1 1; 1 1; 1 1];
tcolor(1,1,1:3) = [1 1 1];
tcolor(1,2,1:3) = [.7 .7 .7];
patch(x,y,z,tcolor)

Notice that each face shares two vertices with the other face (V₁-V₄ and V₃-V₅).

**Specifying Vertices and Faces**

The Vertices property contains the coordinates of each *unique* vertex defining the patch. The Faces property specifies how to connect these vertices to form each face of the patch. For this example, two vertices share the same location so you need to specify only four of the six vertices. Each row contains the x-, y-, and z-coordinates of each vertex.

\[
\text{vert} = [0 1 1; 0 2 1; 1 2 1; 1 1 1];
\]

There are only two faces, defined by connecting the vertices in the order indicated.
To specify the face colors, define a 2-by-3 matrix containing two RGB color definitions.

tcolor = [1 1 1; 0.7 0.7 0.7];

With two faces and two colors, MATLAB can color each face with flat shading. This means you must set the FaceColor property to flat, since the faces/vertices technique is available only as a low-level function call (i.e., only by specifying property name/property value pairs).

Create the patch by specifying the Faces, Vertices, and FaceVertexCData properties as well as the FaceColor property.

patch('Faces', fac, 'Vertices', vert, 'FaceVertexCData', tcolor, 'FaceColor', 'flat');

Specifying only unique vertices and their connection matrix can reduce the size of the data for patches having many faces. See the descriptions of the Faces, Vertices, and FaceVertexCData properties for information on how to define them.

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the end of the Faces matrix...
with NaNs. To define a patch with faces that do not close, add one or more NaNs to the row in the Vertices matrix that defines the vertex you do not want connected.

Creating Pictures with Multiple Patch Objects

The following example recreates a popular puzzle game using 3- and 4-sided patch objects. The first method specifies the x-, y-, and z-coordinates and color data to draw the puzzle pieces in their original square formation. The second method rearranges those pieces into the form of a rabbit, but specifies vertices, the connection matrix, and color data. In each case, the matrices for the alternative specification method are presented beneath the created graphic.

Using x-, y-, and z-coordinates and color data, define first the quadrilateral faces then the triangle face coordinates:

\[
\begin{align*}
    \text{xquad} &= [5,0;10,0;15,5;10,5]; \\
    \text{yquad} &= [15,0;20,10;15,15;10,5]; \\
    \text{zquad} &= \text{ones}(4,2); \\
    \text{xtri} &= [0,10,10,5,0;10,20,20,5,0;20,20,15,10,10]; \\
    \text{ytri} &= [0,10,20,5,10;10,20,20,15,20;0,0,15,10,20]; \\
    \text{ztri} &= \text{zeros}(3,5);
\end{align*}
\]

Create the patch faces, coloring them white:

\[
\begin{align*}
    \text{p1} &= \text{patch}(\text{xtri}, \text{ytri}, \text{ztri}, \text{FaceColor}, \text{w}); \\
    \text{p2} &= \text{patch}(\text{xquad}, \text{yquad}, \text{zquad}, \text{FaceColor}, \text{w}); \\
    \text{set(gca, DataAspectRatio, [1 1 1]);}
\end{align*}
\]
To create this image by specifying vertices and connection matrices, use the following inputs:

```matlab
verts = [0 20 0;10 20 0;20 20 0;15 15 0;5 15 0;... 0 10 0;10 10 0;5 5 0;0 0 0;20 0 0];
facestri = [9 7 10;7 3 10;2 3 4;8 5 7;6 1 2];
facesquad = [2 4 7 5;6 5 8 9];
patch('Faces',facesquad,'Vertices',verts,'facecolor','w')
patch('Faces',facestri,'Vertices',verts,'facecolor','w')
```

To create a white rabbit from the same shapes, use the following inputs to define the vertices and connection matrices:

```matlab
rabvert = [11 28 0;6 28 0;6 23 0;5 23 0;1 23 0;0 23 0;...]
```
5 20 0;0 18 0;5 18 0;5 12 0;2.5 9.5 0;5 7 0;...
5 10 0;15 10 0;15 0 0;10 5 0;10 0 0;7.5 2.5 0]
rabquadface = [5 2 1 3;8 6 4 9];
rabtriface = [7 14 13;14 15;11 10 12;18 16 17;17 16 15];

Draw the patches in white and set the background to black:

```
patch('Faces',rabquadface,'Vertices',rabvert,'FaceColor','w')
patch('Faces',rabtriface,'Vertices',rabvert,'FaceColor','w')
set(gca,'Color','black','DataAspectRatio',[1 1 1]);
```

To create this image using x-, y-, and z-coordinates, use the following inputs:

```
rabxtri = [5,5,2.5,7.5,10;5,15,5,10,10;15,15,5,10,15;];
```
rabytri = [10,10,9.5,2.5,0;20,10,12,5,5;10 0 7 0 0];
rabztri = [0,0,0,0,0;0,0,0,0,0;0,0,0,0,0;];
rabxquad = [0,1;0,6;5,11;5,6;];
rabyquad = [18,23;23,28;23,28;18,23;];
rabzquad = [1,1;1,1;1,1;1,1;];
patch(rabxtri,rabytri,rabztri,'FaceColor','w')
patch(rabxquad,rabyquad,rabzquad,'FaceColor','w')
set(gca,'Color','black','DataAspectRatio',[1 1 1])

Setting Default Properties

You can set default patch properties on the axes, figure, and root object levels:

set(0,'DefaultPatchPropertyName',PropertyValue...)  
set(gcf,'DefaultPatchPropertyName',PropertyValue...)  
set(gca,'DefaultPatchPropertyName',PropertyValue...)  

PropertyName is the name of the patch property and PropertyValue is the value you are specifying. Use set and get to access patch properties.

See Also

area, caxis, fill, fill3, isosurface, surface

“Object Creation” on page 1-101 for related functions

Patch Properties for property descriptions

“Creating 3-D Models with Patches” for examples that use patches
Patch Properties

**Purpose**
Patch properties

**Modifying Properties**
You can set and query graphics object properties in two ways:

- “The Property Editor” is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see “Setting Default Property Values”.

See “Core Graphics Objects” for general information about this type of object.

**Patch Property Descriptions**
This section lists property names along with the type of values each accepts. Curly braces {} enclose default values.

**AlphaDataMapping**

none | {scaled} | direct

*Transparency mapping method.* This property determines how the MATLAB software interprets indexed alpha data. This property can be any of the following:

- **none** — The transparency values of `FaceVertexAlphaData` are between 0 and 1 or are clamped to this range.

- **scaled** — Transform the `FaceVertexAlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values. (`scaled` is the default)

- **direct** — Use the `FaceVertexAlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from 1 to `length(alphamap)`. MATLAB maps values less than 1 to the first alpha value in the alphamap, and values greater than `length(alphamap)`
to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If FaceVertexAlphaData is an array of uint8 integers, then the indexing begins at 0 (i.e., MATLAB maps a value of 0 to the first alpha value in the alphamap).

**AmbientStrength**

scalar >= 0 and <= 1

*Strength of ambient light.* This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes AmbientColor property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the DiffuseStrength and SpecularStrength properties.

**Annotation**

*hg.Annotation* object Read Only

*Control the display of patch objects in legends.* The Annotation property enables you to specify whether this patch object is represented in a figure legend.

Querying the Annotation property returns the handle of an *hg.Annotation* object. The *hg.Annotation* object has a property called LegendInformation, which contains an *hg.LegendEntry* object.

Once you have obtained the *hg.LegendEntry* object, you can set its IconDisplayStyle property to control whether the patch object is displayed in a figure legend:
### IconDisplayStyle Property

<table>
<thead>
<tr>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Represent this patch object in a legend (default)</td>
</tr>
<tr>
<td>off</td>
<td>Do not include this patch object in a legend</td>
</tr>
<tr>
<td>children</td>
<td>Same as on because patch objects do not have children</td>
</tr>
</tbody>
</table>

**Setting the IconDisplayStyle property**

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

**Using the IconDisplayStyle property**

See “Controlling Legends” for more information and examples.

**Selecting which objects to display in legend**

Some graphics functions create multiple objects. For example, `contour3` uses patch objects to create a 3D contour graph. You can use the `Annotation` property set select a subset of the objects for display in the legend.

```matlab
[X,Y] = meshgrid(-2:.1:2);
[Cm hC] = contour3(X.*exp(-X.^2-Y.^2));
hA = get(hC,'Annotation');
hLL = get([hA{:}],'LegendInformation');
% Set the IconDisplayStyle property to display
% the first, fifth, and ninth patch in the legend
set([hLL{:}],'IconDisplayStyle',{...'
    'on','off','off','off','on','off','off','off','on'});
```
% Assign DisplayNames for the three patch
that are displayed in the legend
set(hC([1,5,9]),{'DisplayName'},{'bottom','middle','top'}')
legend show

BackFaceLighting
unlit | lit | {reverselit}

*Face lighting control.* This property determines how faces are lit
when their vertex normals point away from the camera:

- **unlit** — Face is not lit.
- **lit** — Face is lit in normal way.
- **reverselit** — Face is lit as if the vertex pointed towards the
camera.

This property is useful for discriminating between the internal
and external surfaces of an object. See the Using MATLAB
Graphics manual for an example.

BeingDeleted
on | {off} Read Only

*This object is being deleted.* The BeingDeleted property provides
a mechanism that you can use to determine if objects are in
the process of being deleted. MATLAB sets the BeingDeleted
property to *on* when the object’s delete function callback is called
(see the DeleteFcn property) It remains set to *on* while the delete
function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions
that act on a number of different objects. These functions may not
need to perform actions on objects that are going to be deleted,
and therefore, can check the object’s BeingDeleted property
before acting.

BusyAction
cancel | {queue}
Callback routine interruption. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- cancel — Discard the event that attempted to execute a second callback routine.
- queue — Queue the event that attempted to execute a second callback routine until the current callback finishes.

ButtonDownFcn
function handle, cell array containing function handle and additional arguments, or string (not recommended)

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the patch object.

See the figure’s SelectionType property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. You can also use a string that is a valid MATLAB expression or the name of an M-file. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

CData
scalar, vector, or matrix
Patch colors. This property specifies the color of the patch. You can specify color for each vertex, each face, or a single color for the entire patch. The way MATLAB interprets CData depends on the type of data supplied. The data can be numeric values that are scaled to map linearly into the current colormap, integer values that are used directly as indices into the current colormap, or arrays of RGB values. RGB values are not mapped into the current colormap, but interpreted as the colors defined. On true color systems, MATLAB uses the actual colors defined by the RGB triples.

The following two diagrams illustrate the dimensions of CData with respect to the coordinate data arrays, XData, YData, and ZData. The first diagram illustrates the use of indexed color.

The second diagram illustrates the use of true color. True color requires $m$-by-$n$-by-3 arrays to define red, green, and blue components for each color.
Note that if CData contains NaNs, MATLAB does not color the faces.

See also the Faces, Vertices, and FaceVertexCData properties for an alternative method of patch definition.

**CDataMapping**

{scaled} | direct

*Direct or scaled color mapping.* This property determines how MATLAB interprets indexed color data used to color the patch. (If you use true color specification for CData or FaceVertexCData, this property has no effect.)

- **scaled** — Transform the color data to span the portion of the colormap indicated by the axes CLim property, linearly mapping data values to colors. See the caxis command for more information on this mapping.
- **direct** — Use the color data as indices directly into the colormap. When not scaled, the data are usually integer values
ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

**Children**

matrix of handles

Always the empty matrix; patch objects have no children.

**Clipping**

{on} | off

*Clipping to axes rectangle.* When Clipping is on, MATLAB does not display any portion of the patch outside the axes rectangle.

**CreateFcn**

string or function handle

*Callback routine executed during object creation.* This property defines a callback routine that executes when MATLAB creates a patch object. You must define this property as a default value for patches or in a call to the `patch` function that creates a new object.

For example, the following statement creates a patch (assuming x, y, z, and c are defined), and executes the function referenced by the function handle `@myCreateFcn`.

```
patch(x,y,z,c,'CreateFcn',@myCreateFcn)
```

MATLAB executes the create function after setting all properties for the patch created. Setting this property on an existing patch object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcb0`.
See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

DeleteFcn
string or function handle

Delete patch callback routine. A callback routine that executes when you delete the patch object (e.g., when you issue a delete command or clear the axes (cla) or figure (clf) containing the patch). MATLAB executes the routine before deleting the object’s properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

DiffuseStrength
scalar >= 0 and <= 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the patch. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the patch object. See the AmbientStrength and SpecularStrength properties.

DisplayName
string (default is empty string)

String used by legend for this patch object. The legend function uses the string defined by the DisplayName property to label this patch object in the legend.
- If you specify string arguments with the `legend` function, `DisplayName` is set to this patch object’s corresponding string and that string is used for the legend.

- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where `n` is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.

- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.

- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.

- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

**EdgeAlpha**

`{scalar = 1} | flat | interp`

_Transparency of the edges of patch faces._ This property can be any of the following:

- **scalar** — A single non-NaN scalar value between 0 and 1 that controls the transparency of all the edges of the object. 1 (the default) means fully opaque and 0 means completely transparent.

- **flat** — The alpha data (`FaceVertexAlphaData`) of each vertex controls the transparency of the edge that follows it.

- **interp** — Linear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determines the transparency of the edge.
Note that you cannot specify flat or interp EdgeAlpha without first setting FaceVertexAlphaData to a matrix containing one alpha value per face (flat) or one alpha value per vertex (interp).

**EdgeColor**

\{ColorSpec\} | none | flat | interp

*Color of the patch edge.* This property determines how MATLAB colors the edges of the individual faces that make up the patch.

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default edge color is black. See ColorSpec for more information on specifying color.
- **none** — Edges are not drawn.
- **flat** — The color of each vertex controls the color of the edge that follows it. This means flat edge coloring is dependent on the order in which you specify the vertices:

\[\]

- **interp** — Linear interpolation of the CData or FaceVertexCData values at the vertices determines the edge color.

**EdgeLighting**

\{none\} | flat | gouraud | phong

*Algorithm used for lighting calculations.* This property selects the algorithm used to calculate the effect of light objects on patch edges. Choices are
Patch Properties

- **none** — Lights do not affect the edges of this object.
- **flat** — The effect of light objects is uniform across each edge of the patch.
- **gouraud** — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- **phong** — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**EraseMode**

{normal} | none | xor | background

_Erase mode._ This property controls the technique MATLAB uses to draw and erase patch objects. Alternative erase modes are useful in creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase the patch when it is moved or destroyed. While the object is still visible on the screen after erasing with EraseMode none, you cannot print it because MATLAB stores no information about its former location.
- **xor** — Draw and erase the patch by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the patch does not damage the color of the objects behind it. However, patch color depends on the color of the screen behind it and is correctly colored only when over the axes background.
** Patch Properties 

Color, or the figure background Color if the axes Color is set to none.

- **background** — Erase the patch by drawing it in the axes background Color, or the figure background Color if the axes Color is set to none. This damages objects that are behind the erased patch, but the patch is always properly colored.

**Printing with Nonnormal Erase Modes**

MATLAB always prints figures as if the `EraseMode` of all objects is normal. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., perform an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing nonnormal mode objects.

FaceAlpha

```matlab
{scalar = 1} | flat | interp
```

*Transparency of the patch face.* This property can be any of the following:

- **A scalar** — A single non-NaN value between 0 and 1 that controls the transparency of all the faces of the object. 1 (the default) means fully opaque and 0 means completely transparent (invisible).

- **flat** — The values of the alpha data (`FaceVertexAlphaData`) determine the transparency for each face. The alpha data at the first vertex determines the transparency of the entire face.

- **interp** — Bilinear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determines the transparency of each face.
Note that you cannot specify flat or interp FaceAlpha without first setting FaceVertexAlphaData to a matrix containing one alpha value per face (flat) or one alpha value per vertex (interp).

**FaceColor**

{ColorSpec} | none | flat | interp

*Color of the patch face.* This property can be any of the following:

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See ColorSpec for more information on specifying color.
- **none** — Do not draw faces. Note that edges are drawn independently of faces.
- **flat** — The CData or FaceVertexCData property must contain one value per face and determines the color for each face in the patch. The color data at the first vertex determines the color of the entire face.
- **interp** — Bilinear interpolation of the color at each vertex determines the coloring of each face. The CData or FaceVertexCData property must contain one value per vertex.

**FaceLighting**

{none} | flat | gouraud | phong

*Algorithm used for lighting calculations.* This property selects the algorithm used to calculate the effect of light objects on patch faces. Choices are

- **none** — Lights do not affect the faces of this object.
- **flat** — The effect of light objects is uniform across the faces of the patch. Select this choice to view faceted objects.
- **gouraud** — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
Patch Properties

- **phong** — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**Faces**

m-by-n matrix

*Vertex connection defining each face.* This property is the connection matrix specifying which vertices in the **Vertices** property are connected. The **Faces** matrix defines m faces with up to n vertices each. Each row designates the connections for a single face, and the number of elements in that row that are not NaN defines the number of vertices for that face.

The **Faces** and **Vertices** properties provide an alternative way to specify a patch that can be more efficient than using x, y, and z coordinates in most cases. For example, consider the following patch. It is composed of eight triangular faces defined by nine vertices.

The corresponding **Faces** and **Vertices** properties are shown to the right of the patch. Note how some faces share vertices with
other faces. For example, the fifth vertex (V5) is used six times, once each by faces one, two, and three and six, seven, and eight. Without sharing vertices, this same patch requires 24 vertex definitions.

**FaceVertexAlphaData**

m-by-1 matrix

*Face and vertex transparency data.* The FaceVertexAlphaData property specifies the transparency of patches that have been defined by the Faces and Vertices properties. The interpretation of the values specified for FaceVertexAlphaData depends on the dimensions of the data.

FaceVertexAlphaData can be one of the following:

- A single value, which applies the same transparency to the entire patch. The FaceAlpha property must be set to flat.
- An m-by-1 matrix (where m is the number of rows in the Faces property), which specifies one transparency value per face. The FaceAlpha property must be set to flat.
- An m-by-1 matrix (where m is the number of rows in the Vertices property), which specifies one transparency value per vertex. The FaceAlpha property must be set to interp.

The AlphaDataMapping property determines how MATLAB interprets the FaceVertexAlphaData property values.

**FaceVertexCData**

matrix

*Face and vertex colors.* The FaceVertexCData property specifies the color of patches defined by the Faces and Vertices properties. You must also set the values of the FaceColor, EdgeColor, MarkerFaceColor, or MarkerEdgeColor appropriately. The interpretation of the values specified for FaceVertexCData depends on the dimensions of the data.
For indexed colors, FaceVertexCData can be

- A single value, which applies a single color to the entire patch
- An \( n \)-by-1 matrix, where \( n \) is the number of rows in the Faces property, which specifies one color per face
- An \( n \)-by-1 matrix, where \( n \) is the number of rows in the Vertices property, which specifies one color per vertex

For true colors, FaceVertexCData can be

- A 1-by-3 matrix, which applies a single color to the entire patch
- An \( n \)-by-3 matrix, where \( n \) is the number of rows in the Faces property, which specifies one color per face
- An \( n \)-by-3 matrix, where \( n \) is the number of rows in the Vertices property, which specifies one color per vertex

The following diagram illustrates the various forms of the FaceVertexCData property for a patch having eight faces and nine vertices. The CDataMapping property determines how MATLAB interprets the FaceVertexCData property when you specify indexed colors.
HandleVisibility

{on} | callback | off

Control access to object’s handle by command-line users and GUIs. This property determines when an object’s handle is visible in its parent’s list of children. HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle’s visibility is restricted using `callback` or `off`, the object’s handle does not appear in its parent’s `Children` property, figures do not appear in the root’s `CurrentFigure` property, objects do not appear in the root’s `CallbackObject` property or in the figure’s `CurrentObject` property, and axes do not appear in their parent’s `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object’s handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

**HitTest**

| {on} | off |

*Selectable by mouse click.* `HitTest` determines if the patch can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the patch. If `HitTest` is `off`, clicking the patch selects the object below it (which may be the axes containing it).

**Interruptible**

| {on} | off |
**Callback routine interruption mode.** The Interruptible property controls whether a patch callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the ButtonDownFcn are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback routine only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

**LineStyle**

{-} | -- | : | -. | none

*Edge linestyle.* This property specifies the line style of the patch edges. The following table lists the available line styles.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>--</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

You can use LineStyle none when you want to place a marker at each point but do not want the points connected with a line (see the Marker property).

**LineWidth**

scalar

*Edge line width.* The width, in points, of the patch edges (1 point = \(\frac{1}{72}\) inch). The default LineWidth is 0.5 points.

**Marker**

character (see table)
**Marker symbol.** The **Marker** property specifies marks that locate vertices. You can set values for the **Marker** property independently from the **LineStyle** property. The following tables lists the available markers.

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>o</td>
<td>Circle</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

**MarkerEdgeColor**

ColorSpec | none | {auto} | flat

**Marker edge color.** The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- **ColorSpec** — Defines the color to use.
- none — Specifies no color, which makes nonfilled markers invisible.
Patch Properties

- auto — Sets MarkerEdgeColor to the same color as the EdgeColor property.

MarkerFaceColor
ColorSpec | {none} | auto | flat

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- ColorSpec — Defines the color to use.
- none — Makes the interior of the marker transparent, allowing the background to show through.
- auto — Sets the fill color to the axes color, or the figure color, if the axes Color property is set to none.

MarkerSize
size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 of the specified size.

NormalMode
{auto} | manual

MATLAB generated or user-specified normal vectors. When this property is auto, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to manual and does not generate its own data. See also the VertexNormals property.

Parent
handle of axes, hggroup, or hgtransform
**Parent of patch object.** This property contains the handle of the patch object's parent. The parent of a patch object is the axes, hggroup, or hgtransform object that contains it.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

**Selected**

*on | {off}*

*Is object selected?* When this property is on, MATLAB displays selection handles or a dashed box (depending on the number of faces) if the `SelectionHighlight` property is also on. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**SelectionHighlight**

*{on} | off*

*Objects are highlighted when selected.* When the `Selected` property is on, MATLAB indicates the selected state by

- Drawing handles at each vertex for a single-faced patch
- Drawing a dashed bounding box for a multifaced patch

When `SelectionHighlight` is off, MATLAB does not draw the handles.

**SpecularColorReflectance**

*scalar in the range 0 to 1*

*Color of specularly reflected light.* When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color of the light source (i.e., the light object `Color` property). The proportions vary linearly for values in between.
Patch Properties

SpecularExponent
scalar >= 1

_Harshness of specular reflection_. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

SpecularStrength
scalar >= 0 and <= 1

_Intensity of specular light_. This property sets the intensity of the specular component of the light falling on the patch. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the patch object. See the AmbientStrength and DiffuseStrength properties.

Tag
string

_User-specified object label_. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines.

For example, suppose you use patch objects to create borders for a group of uicontrol objects and want to change the color of the borders in a uicontrol’s callback routine. You can specify a Tag with the patch definition

```
patch(X,Y,'k','Tag','PatchBorder')
```

Then use findobj in the uicontrol’s callback routine to obtain the handle of the patch and set its FaceColor property.

```
set(findobj('Tag','PatchBorder'),'FaceColor','w')
```
Type

    string (read only)

*Class of the graphics object.* For patch objects, **Type** is always the string `'patch'`.

**UIContextMenu**

    handle of a uicontextmenu object

*Associate a context menu with the patch.* Assign this property the handle of a uicontextmenu object created in the same figure as the patch. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the patch.

**UserData**

    matrix

*User-specified data.* Any matrix you want to associate with the patch object. MATLAB does not use this data, but you can access it using `set` and `get`.

**VertexNormals**

    matrix

*Surface normal vectors.* This property contains the vertex normals for the patch. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

**Vertices**

    matrix

*Vertex coordinates.* A matrix containing the x-, y-, z-coordinates for each vertex. See the **Faces** property for more information.

**Visible**

    {on}  |  off
Patch Properties

Patch object visibility. By default, all patches are visible. When set to off, the patch is not visible, but still exists, and you can query and set its properties.

XData
vector or matrix

X-coordinates. The x-coordinates of the patch vertices. If XData is a matrix, each column represents the x-coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

YData
vector or matrix

Y-coordinates. The y-coordinates of the patch vertices. If YData is a matrix, each column represents the y-coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

ZData
vector or matrix

Z-coordinates. The z-coordinates of the patch vertices. If ZData is a matrix, each column represents the z-coordinates of a single face of the patch. In this case, XData, YData, and ZData must have the same dimensions.

See Also
patch
Purpose
View or change search path

GUI
As an alternative to the path function, use the Set Path dialog box.

Alternatives

Syntax
path
path('newpath')
path(path,'newpath')
path('newpath',path)
p = path...

Description
path displays the current MATLAB search path. The search path is stored in the file pathdef.m.

path('newpath') changes the search path to newpath, where newpath is a string array of directories.

path(path,'newpath') adds the newpath directory to the bottom of the current search path. If newpath is already on the path, then path(path,'newpath') moves newpath to the end of the path.

path('newpath',path) adds the newpath directory to the top of the current search path. If newpath is already on the path, then path('newpath', path) moves newpath to the beginning of the path.

p = path(...) returns the specified path in string variable p.

Examples
To display the search path, run

    path

MATLAB displays, for example

    MATLABPATH

    H:\My Documents\MATLAB
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\lang
Add a new directory to the search path on Microsoft Windows platforms.

    path(path,'c:/tools/goodstuff')

Add a new directory to the search path on UNIX\(^{21}\) platforms.

    path(path,'/home/tools/goodstuff')

See Also

addpath, cd, dir, genpath, matlabroot, partialpath, pathsep, pathtool, rehash, restddefualtpath, rmpath, savepath, startup, userpath, what

Search Path in the MATLAB Desktop Tools and Development Environment documentation, especially “Programmatically Working with the Search Path”.

\(^{21}\) UNIX is a registered trademark of The Open Group in the United States and other countries.
**Purpose**  
Save current search path to `pathdef.m` file

**Syntax**  
`path2rc`

**Description**  
`path2rc` runs `savepath`. The `savepath` function is replacing `path2rc`. Use `savepath` instead of `path2rc` and replace instances of `path2rc` with `savepath`. 
**pathsep**

**Purpose**  
Path separator for current platform

**Syntax**  
c = pathsep

**Description**  
c = pathsep returns the path separator character for this platform. The path separator is the character that separates directories in the pathdef.m file. The characters is a semicolon (;). For versions of MATLAB software prior to version 7.7 (R2008b), the character on UNIX\(^{22}\) platforms was a colon (:). Use pathsep when you need to work with the path file content programmatically, as a string.

**See Also**  
fileparts, filesep, fullfile, partialpath, path

22. UNIX is a registered trademark of The Open Group in the United States and other countries.
| **Purpose** | Open Set Path dialog box to view and change search path |
| **GUI Alternatives** | As an alternative to the `pathtool` function, select **File > Set Path** in the MATLAB desktop. |
| **Syntax** | `pathtool` |
| **Description** | `pathtool` opens the **Set Path** dialog box, a graphical user interface you use to view and modify the search path MATLAB uses. |
See Also

addpath, cd, dir, genpath, matlabroot, partialpath, path, pathsep, rehash, restoredefaultpath, rmpath, savepath, startup, what

Search Path topics, including “Overview of Viewing and Changing the Search Path”, in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**

Halt execution temporarily

**Syntax**

pause
pause(n)
pause on
pause off
pause query
state = pause('query')
oldstate = pause(newstate)

**Description**

`pause`, by itself, causes the currently executing M-file to stop and wait for you to press any key before continuing. Pausing must be enabled for this to take effect. (See `pause on`, below). `pause` without arguments also blocks execution of Simulink models, but not repainting of them.

`pause(n)` pauses execution for $n$ seconds before continuing, where $n$ can be any nonnegative real number. The resolution of the clock is platform specific. A fractional pause of 0.01 seconds should be supported on most platforms. Pausing must be enabled for this to take effect.

Typing `pause(inf)` puts you into an infinite loop. To return to the MATLAB prompt, type `Ctrl+C`.

`pause on` enables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. Pausing remains enabled until you enter `pause off` in your M-file or at the command line.

`pause off` disables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. This allows normally interactive scripts to run unattended. Pausing remains disabled until you enter `pause on` in your M-file or at the command line, or start a new MATLAB session.

`pause query` displays 'on' if pausing is currently enabled. Otherwise, it displays 'off'.

`state = pause('query')` returns 'on' in character array `state` if pausing is currently enabled. Otherwise, the value of `state` is 'off'.

2-2745
oldstate = pause(newstate), enables or disables pausing, depending on the 'on' or 'off' value in newstate, and returns the former setting (also either 'on' or 'off') in character array oldstate.

Remarks

While MATLAB is paused, the following continue to execute:

- Repainting of figure windows, Simulink block diagrams, and Java windows
- HG callbacks from figure windows
- Event handling from Java windows

See Also

keyboard, input, drawnow
**Purpose**
Set or query plot box aspect ratio

**Syntax**
```python
pbaspect
pbaspect([aspect_ratio])
pbaspect('mode')
pbaspect('auto')
pbaspect('manual')
pbaspect(axes_handle,...)
```

**Description**
The plot box aspect ratio determines the relative size of the x-, y-, and z-axes.

`pbaspect` with no arguments returns the plot box aspect ratio of the current axes.

`pbaspect([aspect_ratio])` sets the plot box aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the x-, y-, and z-axes size. For example, a value of [1 1 1] (the default) means the plot box is a cube (although with stretch-to-fill enabled, it may not appear as a cube). See Remarks.

`pbaspect('mode')` returns the current value of the plot box aspect ratio mode, which can be either `auto` (the default) or `manual`. See Remarks.

`pbaspect('auto')` sets the plot box aspect ratio mode to `auto`.

`pbaspect('manual')` sets the plot box aspect ratio mode to `manual`.

`pbaspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. If you do not specify an axes handle, `pbaspect` operates on the current axes.

**Remarks**
`pbaspect` sets or queries values of the axes object `PlotBoxAspectRatio` and `PlotBoxAspectRatioMode` properties.

When the plot box aspect ratio mode is `auto`, the MATLAB software sets the ratio to [1 1 1], but may change it to accommodate manual settings of the data aspect ratio, camera view angle, or axis limits. See the axes `DataAspectRatio` property for a table listing the interactions between various properties.
Setting a value for the plot box aspect ratio or setting the plot box aspect ratio mode to \texttt{manual} disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the plot box aspect ratio to its current value,

\begin{verbatim}
 pbaspect(pbaspect)
\end{verbatim}

can cause a change in the way the graphs look. See the Remarks section of the \texttt{axes} reference description, “Axes Aspect Ratio Properties” in the 3-D Visualization manual, and “Setting Aspect Ratio” in the MATLAB Graphics manual for a discussion of stretch-to-fill.

**Examples**

The following surface plot of the function \( z = xe^{(-x^2 - y^2)} \) is useful to illustrate the plot box aspect ratio. First plot the function over the range \(-2 \leq x \leq 2, -2 \leq y \leq 2,\)

\begin{verbatim}
 [x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
\end{verbatim}
Querying the plot box aspect ratio shows that the plot box is square.

\begin{verbatim}
  pbaspect
  ans =
      1   1   1
\end{verbatim}

It is also interesting to look at the data aspect ratio selected by MATLAB.

\begin{verbatim}
  daspect
  ans =
      4   4   1
\end{verbatim}

To illustrate the interaction between the plot box and data aspect ratios, set the data aspect ratio to \([1 \ 1 \ 1]\) and again query the plot box aspect ratio.

\begin{verbatim}
  daspect([1 1 1])
\end{verbatim}
pbaspect
ans =
    4    4    1

The plot box aspect ratio has changed to accommodate the specified data aspect ratio. Now suppose you want the plot box aspect ratio to be [1 1 1] as well.

pbaspect([1 1 1])
Notice how MATLAB changed the axes limits because of the constraints introduced by specifying both the plot box and data aspect ratios.

You can also use `pbaspect` to disable stretch-to-fill. For example, displaying two subplots in one figure can give surface plots a squashed appearance. Disabling stretch-to-fill,

```matlab
upper_plot = subplot(211); surf(x,y,z) lower_plot = subplot(212); surf(x,y,z) pbaspect(upper_plot,'manual')
```
See Also

axis, daspect, xlim, ylim, zlim

The axes properties DataAspectRatio, PlotBoxAspectRatio, XLim, YLim, ZLim

Setting Aspect Ratio in the MATLAB Graphics manual

Axes Aspect Ratio Properties in the 3-D Visualization manual
Purpose

Preconditioned conjugate gradients method

Syntax

\[
x = \text{pcg}(A,b)
\]
\[
\text{pcg}(A,b,\text{tol})
\]
\[
\text{pcg}(A,b,\text{tol,\text{maxit}})
\]
\[
\text{pcg}(A,b,\text{tol,\text{maxit}},M)
\]
\[
\text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2)
\]
\[
\text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2,x0)
\]
\[
[x,\text{flag}] = \text{pcg}(A,b,\ldots)
\]
\[
[x,\text{flag,relres}] = \text{pcg}(A,b,\ldots)
\]
\[
[x,\text{flag,relres,iter}] = \text{pcg}(A,b,\ldots)
\]
\[
[x,\text{flag,relres,iter,resvec}] = \text{pcg}(A,b,\ldots)
\]

Description

\[
x = \text{pcg}(A,b)
\]

attempts to solve the system of linear equations \(A*x=b\) for \(x\). The \(n\)-by-\(n\) coefficient matrix \(A\) must be symmetric and positive definite, and should also be large and sparse. The column vector \(b\) must have length \(n\). \(A\) can be a function handle \(\text{afun}\) such that \(\text{afun}(x)\) returns \(A*x\). See Function Handles in the MATLAB Programming documentation for more information.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function \(\text{afun}\), as well as the preconditioner function \(\text{mfun}\) described below, if necessary.

If \(\text{pcg}\) converges, a message to that effect is displayed. If \(\text{pcg}\) fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual \(\|b-A*x\|/\|b\|\) and the iteration number at which the method stopped or failed.

\(\text{pcg}(A,b,\text{tol})\) specifies the tolerance of the method. If \(\text{tol}\) is [], then \(\text{pcg}\) uses the default, \(1e-6\).

\(\text{pcg}(A,b,\text{tol,\text{maxit}})\) specifies the maximum number of iterations. If \(\text{maxit}\) is [], then \(\text{pcg}\) uses the default, \(\min(n,20)\).

\(\text{pcg}(A,b,\text{tol,\text{maxit}},M)\) and \(\text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2)\) use symmetric positive definite preconditioner \(M\) or \(M = M1*M2\) and
effectively solve the system \( \text{inv}(M) * A * x = \text{inv}(M) * b \) for \( x \). If \( M \) is []
then \text{pcg} applies no preconditioner. \( M \) can be a function handle \text{mfun}
such that \text{mfun}(x) returns \( M \backslash x \).

\text{pcg}(A,b,\text{tol},\text{maxit},M1,M2,x0) \) specifies the initial guess. If \( x0 \) is [],
then \text{pcg} uses the default, an all-zero vector.

\( [x,\text{flag}] = \text{pcg}(A,b,\ldots) \) also returns a convergence flag.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\text{pcg} converged to the desired tolerance \text{tol} within \text{maxit} iterations.</td>
</tr>
<tr>
<td>1</td>
<td>\text{pcg} iterated \text{maxit} times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>Preconditioner ( M ) was ill-conditioned.</td>
</tr>
<tr>
<td>3</td>
<td>\text{pcg} stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during \text{pcg} became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

Whenever \( \text{flag} \) is not 0, the solution \( x \) returned is that with minimal
norm residual computed over all the iterations. No messages are
displayed if the \( \text{flag} \) output is specified.

\( [x,\text{flag},\text{relres}] = \text{pcg}(A,b,\ldots) \) also returns the relative residual
\( \text{norm}(b-A*x)/\text{norm}(b) \). If \( \text{flag} \) is 0, \( \text{relres} \leq \text{tol} \).

\( [x,\text{flag},\text{relres},\text{iter}] = \text{pcg}(A,b,\ldots) \) also returns the iteration
number at which \( x \) was computed, where 0 \( \leq \text{iter} \leq \text{maxit} \).

\( [x,\text{flag},\text{relres},\text{iter},\text{resvec}] = \text{pcg}(A,b,\ldots) \) also returns a vector
of the residual norms at each iteration including \( \text{norm}(b-A*x0) \).

**Examples**

**Example 1**

```matlab
n1 = 21;
A = gallery('moler',n1);
b1 = A*ones(n1,1);
tol = 1e-6;
```
maxit = 15;
M = diag([10:-1:1 1:10]);
[x1,flag1,rr1,iter1,rv1] = pcg(A,b1,tol,maxit,M);

Alternatively, you can use the following parameterized matrix-vector product function afun in place of the matrix A:

    afun = @(x,n)gallery('moler',n)*x;
n2 = 21;
b2 = afun(ones(n2,1),n2);
[x2,flag2,rr2,iter2,rv2] = pcg(@(x)afun(x,n2),b2,tol,maxit,M);

Example 2

    A = delsq(numgrid('C',25));
b = ones(length(A),1);
[x,flag] = pcg(A,b)

flag is 1 because pcg does not converge to the default tolerance of 1e-6 within the default 20 iterations.

    R = cholinc(A,1e-3);
[x2,flag2,relres2,iter2,resvec2] = pcg(A,b,1e-8,10,R',R)

flag2 is 0 because pcg converges to the tolerance of 1.2e-9 (the value of relres2) at the sixth iteration (the value of iter2) when preconditioned by the incomplete Cholesky factorization with a drop tolerance of 1e-3.

resvec2(1) = norm(b) and resvec2(7) = norm(b-A*x2). You can follow the progress of pcg by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

    semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
See Also
bicg, bicgstab, cgs, cholinc, gmres, lsqr, minres, qmr, symmlq

function_handle (@), mldivide (\)

References
**Purpose** Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)

**Syntax**

\[
y_i = \text{pchip}(x,y,xi) \\
\text{pp} = \text{pchip}(x,y)
\]

**Description**

\(y_i = \text{pchip}(x,y,xi)\) returns vector \(y_i\) containing elements corresponding to the elements of \(xi\) and determined by piecewise cubic interpolation within vectors \(x\) and \(y\). The vector \(x\) specifies the points at which the data \(y\) is given. If \(y\) is a matrix, then the interpolation is performed for each column of \(y\) and \(y_i\) is \(\text{length}(xi)\)-by-\(\text{size}(y,2)\).

\(\text{pp} = \text{pchip}(x,y)\) returns a piecewise polynomial structure for use by \(\text{ppval}\). \(x\) can be a row or column vector. \(y\) is a row or column vector of the same length as \(x\), or a matrix with \(\text{length}(x)\) columns.

\(\text{pchip}\) finds values of an underlying interpolating function \(P(x)\) at intermediate points, such that:

- On each subinterval \(x_k \leq x \leq x_{k+1}\), \(P(x)\) is the cubic Hermite interpolant to the given values and certain slopes at the two endpoints.

- \(P(x)\) interpolates \(y\), i.e., \(P(x_j) = y_j\), and the first derivative \(P'(x)\) is continuous. \(P''(x)\) is probably not continuous; there may be jumps at the \(x_j\).

- The slopes at the \(x_j\) are chosen in such a way that \(P(x)\) preserves the shape of the data and respects monotonicity. This means that, on intervals where the data are monotonic, so is \(P(x)\); at points where the data has a local extremum, so does \(P(x)\).

**Note** If \(y\) is a matrix, \(P(x)\) satisfies the above for each column of \(y\).
Remarks

spline constructs $S(x)$ in almost the same way pchip constructs $P(x)$. However, spline chooses the slopes at the $x_j$ differently, namely to make even $S''(x)$ continuous. This has the following effects:

- spline produces a smoother result, i.e. $S''(x)$ is continuous.
- spline produces a more accurate result if the data consists of values of a smooth function.
- pchip has no overshoots and less oscillation if the data are not smooth.
- pchip is less expensive to set up.
- The two are equally expensive to evaluate.

Examples

```matlab
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
t = -3:.01:3;
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'o',t,p,'-',t,s,'-.')
legend('data','pchip','spline',4)
```
See Also

interp1, spline, ppval

References


**Purpose**
Create protected M-file (P-file)

**Syntax**
```
pcode fun
pcode *.m
pcode fun1 fun2 ...
pcode... -inplace
```

**Description**
pcode fun obfuscates (i.e., shrouds) M-file fun.m for the purpose of protecting its proprietary source code. The encrypted M-code is written to P-file fun.p in the current directory. The original M-file can be anywhere on the search path.

If the input file resides within a package and/or class directory, then the same package and class directories are applied to the output file. See example 2, below.

pcode *.m creates P-files for all the M-files in the current directory.

pcode fun1 fun2 ... creates P-files for the listed functions.

pcode... -inplace creates P-files in the same directory as the M-files. An error occurs if the files cannot be created.

See “Creating P-Code Files” in the MATLAB Programming Fundamentals documentation for more information.

**Examples**

**Example 1 – PCoding Multiple Files**
Convert selected M-files from the sparfun directory into pcode files:

```matlab
dir(fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m'))
   ..   sprand.m sprandn.m sprandsym.m sprank.m

cd C:\work\pcodetest
pcode(fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m'))

dir C:\work\pcodetest
   ..   sprand.m sprandn.m sprandsym.m sprank.m
```
. . sprand.p sprandn.p sprandsym.p sprank.p

**Example 2 – Parsing Files That Belong to a Package and/or Class**

This example takes an input file that is part of a package and class, and generates a pcode file for it in a separate directory. M-file `test.m` resides in the following package and class directory:

```
C:\work\+mypkg\@char\test.m
```

Set your current working directory to empty directory `math\pcodetest`. This is where you will generate the pcode file. This directory has no package or class structure associated with it at this time:

```
cd C:\math\pcodetest
dir
  .. ..
```

Generate pcode for `test.m`. Because the input file is part of a package and class, MATLAB creates directories `+mypkg` and `@char` so that the output file belongs to the same:

```
pcode C:\work\+mypkg\@char\test.m
dir('C:\math\pcodetest\+mypkg\@char')
  .. ..  test.p
```

**Example 3 – PCoding In Place**

When you generate a pcode file `inplace`, MATLAB writes the output file to the same directory as the input file:

```
pcode C:\work\+mypkg\@char\test.m  -inplace
dir C:\work\+mypkg\@char
  .. ..  test.m  test.p
```

**See Also**

depfun, depdir,
**Purpose**

Pseudocolor (checkerboard) plot

**GUI Alternatives**

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

```matlab
pcolor(C)
pcolor(X,Y,C)
pcolor(axes_handles,...)
h = pcolor(...)
```

**Description**

A pseudocolor plot is a rectangular array of cells with colors determined by C. MATLAB creates a pseudocolor plot using each set of four adjacent points in C to define a surface rectangle (i.e., cell).

The default shading is faceted, which colors each cell with a single color. The last row and column of C are not used in this case. With shading interp, each cell is colored by bilinear interpolation of the colors at its four vertices, using all elements of C.

The minimum and maximum elements of C are assigned the first and last colors in the colormap. Colors for the remaining elements in C are determined by a linear mapping from value to colormap element.

`pcolor(C)` draws a pseudocolor plot. The elements of C are linearly mapped to an index into the current colormap. The mapping from C to the current colormap is defined by colormap and caxis.

`pcolor(X,Y,C)` draws a pseudocolor plot of the elements of C at the locations specified by X and Y. The plot is a logically rectangular, two-dimensional grid with vertices at the points \[ X(i,j), Y(i,j) \]. X and Y are vectors or matrices that specify the spacing of the grid lines. If
X and Y are vectors, X corresponds to the columns of C and Y corresponds to the rows. If X and Y are matrices, they must be the same size as C.

`pcolor(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (gca).

`h = pcolor(...)` returns a handle to a surface graphics object.

**Remarks**

A pseudocolor plot is a flat surface plot viewed from above.

`pcolor(X,Y,C)` is the same as viewing `surf(X,Y,zeros(size(X)),C)` using `view([0 90])`.

When you use shading faceted or shading flat, the constant color of each cell is the color associated with the corner having the smallest x-y coordinates. Therefore, `C(i,j)` determines the color of the cell in the ith row and jth column. The last row and column of C are not used.

When you use shading interp, each cell’s color results from a bilinear interpolation of the colors at its four vertices, and all elements of C are used.

**Examples**

A Hadamard matrix has elements that are +1 and -1. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix.

```matlab
pcolor(hadamard(20))
colormap(gray(2))
axis ij
axis square
```
A simple color wheel illustrates a polar coordinate system.

```matlab
n = 6;
r = (0:n)'/n;
theta = pi*(-n:n)/n;
X = r*cos(theta);
Y = r*sin(theta);
C = r*cos(2*theta);
pcolor(X,Y,C)
axis equal tight
```
Algorithm

The number of vertex colors for `pcolor(C)` is the same as the number of cells for `image(C)`. `pcolor` differs from `image` in that `pcolor(C)` specifies the colors of vertices, which are scaled to fit the colormap; changing the axes `clim` property changes this color mapping. `image(C)` specifies the colors of cells and directly indexes into the colormap without scaling. Additionally, `pcolor(X,Y,C)` can produce parametric grids, which is not possible with `image`.

See Also

caxis, image, mesh, shading, surf, view
**Purpose**
Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D

**Syntax**

```matlab
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>A parameter corresponding to the symmetry of the problem. ( m ) can be slab = 0, cylindrical = 1, or spherical = 2.</td>
</tr>
<tr>
<td>pdefun</td>
<td>A handle to a function that defines the components of the PDE.</td>
</tr>
<tr>
<td>icfun</td>
<td>A handle to a function that defines the initial conditions.</td>
</tr>
<tr>
<td>bcfun</td>
<td>A handle to a function that defines the boundary conditions.</td>
</tr>
<tr>
<td>xmesh</td>
<td>A vector ([x0, x1, ..., xn]) specifying the points at which a numerical solution is requested for every value in ( tspan ). The elements of ( xmesh ) must satisfy ( x0 &lt; x1 &lt; ... &lt; xn ). The length of ( xmesh ) must be ( &gt;= 3 ).</td>
</tr>
<tr>
<td>tspan</td>
<td>A vector ([t0, t1, ..., tf]) specifying the points at which a solution is requested for every value in ( xmesh ). The elements of ( tspan ) must satisfy ( t0 &lt; t1 &lt; ... &lt; tf ). The length of ( tspan ) must be ( &gt;= 3 ).</td>
</tr>
<tr>
<td>options</td>
<td>Some options of the underlying ODE solver are available in \texttt{pdepe}: \texttt{RelTol}, \texttt{AbsTol}, \texttt{NormControl}, \texttt{InitialStep}, and \texttt{MaxStep}. In most cases, default values for these options provide satisfactory solutions. See \texttt{odeset} for details.</td>
</tr>
</tbody>
</table>

**Description**

\( \text{sol} = \text{pdepe}(m, \text{pdefun}, \text{icfun}, \text{bcfun}, \text{xmesh}, \text{tspan}) \) solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable \( x \) and time \( t \). \texttt{pdefun}, \texttt{icfun}, and
bcfun are function handles. See “Function Handles” in the MATLAB Programming documentation for more information. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at times specified in tspan. The pdepe function returns values of the solution on a mesh provided in xmesh.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the functions pdedef, icfun, or bcfun, if necessary.

pdepe solves PDEs of the form:

\[
c(x, t, u, \frac{\partial u}{\partial x}) \frac{\partial u}{\partial t} = x^{-m} \frac{\partial}{\partial x} \left( x^m f(x, t, u, \frac{\partial u}{\partial x}) \right) + s(x, t, u, \frac{\partial u}{\partial x})
\]

The PDEs hold for \( t_0 \leq t \leq t_f \) and \( a \leq x \leq b \). The interval \([a, b]\) must be finite. \( m \) can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If \( m > 0 \), then \( a \) must be \( \geq 0 \).

In Equation 2-2, \( f(x, t, u, \frac{\partial u}{\partial x}) \) is a flux term and \( s(x, t, u, \frac{\partial u}{\partial x}) \) is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix \( c(x, t, u, \frac{\partial u}{\partial x}) \). The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of \( c \) that corresponds to a parabolic equation can vanish at isolated values of \( x \) if those values of \( x \) are mesh points. Discontinuities in \( c \) and/or \( s \) due to material interfaces are permitted provided that a mesh point is placed at each interface.

For \( t = t_0 \) and all \( x \), the solution components satisfy initial conditions of the form
\[ u(x, t_0) = u_0(x) \]  \hfill (2-3)

For all \( t \) and either \( x = a \) or \( x = b \), the solution components satisfy a boundary condition of the form

\[ p(x, t, u) + q(x, t) f(x, t, u, \frac{\partial u}{\partial x}) = 0 \]  \hfill (2-4)

Elements of \( q \) are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux \( f \) rather than \( \frac{\partial u}{\partial x} \). Also, of the two coefficients, only \( p \) can depend on \( u \).

In the call \( \text{sol} = \text{pdepe}(m, \text{pdefun}, \text{icfun}, \text{bcfun}, \text{xmesh}, \text{tspan}) \):

- \( m \) corresponds to \( m \).
- \( \text{xmesh}(1) \) and \( \text{xmesh}(\text{end}) \) correspond to \( a \) and \( b \).
- \( \text{tspan}(1) \) and \( \text{tspan}(\text{end}) \) correspond to \( t_0 \) and \( t_f \).
- \( \text{pdefun} \) computes the terms \( c, f, \) and \( s \) (Equation 2-2). It has the form

\[
[c, f, s] = \text{pdefun}(x, t, u, \text{dudx})
\]

The input arguments are scalars \( x \) and \( t \) and vectors \( u \) and \( \text{dudx} \) that approximate the solution \( u \) and its partial derivative with respect to \( x \), respectively. \( c, f, \) and \( s \) are column vectors. \( c \) stores the diagonal elements of the matrix \( C \) (Equation 2-2).

- \( \text{icfun} \) evaluates the initial conditions. It has the form

\[
\text{u} = \text{icfun}(x)
\]

When called with an argument \( x \), \( \text{icfun} \) evaluates and returns the initial values of the solution components at \( x \) in the column vector \( u \).

- \( \text{bcfun} \) evaluates the terms \( p \) and \( q \) of the boundary conditions (Equation 2-4). It has the form

\[
[pl, ql, pr, qr] = \text{bcfun}(xl, ul, xr, ur, t)
\]
u1 is the approximate solution at the left boundary x1 = a and ur is the approximate solution at the right boundary xR = b. p1 and q1 are column vectors corresponding to p and q evaluated at x1, similarly pr and qr correspond to xR. When m > 0 and a = 0, boundedness of the solution near x = 0 requires that the flux $\hat{f}$ vanish at a = 0. pdepe imposes this boundary condition automatically and it ignores values returned in p1 and q1.

pdepe returns the solution as a multidimensional array sol.
$u_i = u_i = sol(:, :, i)$ is an approximation to the $i$th component of the solution vector $u$. The element $u_i(j, k) = sol(j, k, i)$ approximates $u_i$ at $(t, x) = (tspan(j), xmesh(k))$.

$u_i = sol(j, :, i)$ approximates component $i$ of the solution at time $tspan(j)$ and mesh points $xmesh(:)$. Use pdeval to compute the approximation and its partial derivative $\partial u_i / \partial x$ at points not included in $xmesh$. See pdeval for details.

$\text{sol} = \text{pdepe}(m, \text{pdefun}, \text{icfun}, \text{bcfun}, \text{xmesh}, \text{tspan}, \text{options})$ solves as above with default integration parameters replaced by values in options, an argument created with the odeset function. Only some of the options of the underlying ODE solver are available in pdepe: RelTol, AbsTol, NormControl, InitialStep, and MaxStep. The defaults obtained by leaving off the input argument options will generally be satisfactory. See odeset for details.

**Remarks**

- The arrays xmesh and tspan play different roles in pdepe.

  **tspan** – The pdepe function performs the time integration with an ODE solver that selects both the time step and formula dynamically. The elements of tspan merely specify where you want answers and the cost depends weakly on the length of tspan.

  **xmesh** – Second order approximations to the solution are made on the mesh specified in xmesh. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. pdepe does not select the mesh in $x$ automatically. You must provide an appropriate
fixed mesh in xmesh. The cost depends strongly on the length of xmesh. When \( m > 0 \), it is not necessary to use a fine mesh near \( x = 0 \) to account for the coordinate singularity.

- The time integration is done with ode15s. pdepe exploits the capabilities of ode15s for solving the differential-algebraic equations that arise when Equation 2-2 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.

- After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not "consistent" with the discretization, pdepe tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, pdepe can find consistent initial conditions close to the given ones. If pdepe displays a message that it has difficulty finding consistent initial conditions, try refining the mesh.

No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

**Examples**

**Example 1.** This example illustrates the straightforward formulation, computation, and plotting of the solution of a single PDE.

\[
\frac{\pi^2}{2} \frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left( \frac{\partial u}{\partial x} \right)
\]

This equation holds on an interval \( 0 \leq x \leq 1 \) for times \( t \geq 0 \).

The PDE satisfies the initial condition

\[ u(x, 0) = \sin \pi x \]

and boundary conditions
\[ u(0,t) = 0 \]
\[ \pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0 \]

It is convenient to use subfunctions to place all the functions required by pdepe in a single M-file.

```matlab
function pdex1

m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);

sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
% Extract the first solution component as u.
u = sol(:,:,1);

% A surface plot is often a good way to study a solution.
surf(x,t,u)
title('Numerical solution computed with 20 mesh points.')
xlabel('Distance x')
ylabel('Time t')

% A solution profile can also be illuminating.
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')

% ------------------------------------------------------------
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;

% ------------------------------------------------------------
function u0 = pdex1ic(x)
u0 = sin(pi*x);
```

2-2771
In this example, the PDE, initial condition, and boundary conditions are coded in subfunctions `pdex1pde`, `pdex1ic`, and `pdex1bc`.

The surface plot shows the behavior of the solution.

The following plot shows the solution profile at the final value of $t$ (i.e., $t = 2$).
Example 2. This example illustrates the solution of a system of PDEs. The problem has boundary layers at both ends of the interval. The solution changes rapidly for small $t$.

The PDEs are

\[
\frac{\partial u_1}{\partial t} = 0.024 \frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)
\]

\[
\frac{\partial u_2}{\partial t} = 0.170 \frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)
\]

where $F(y) = \exp(5.73y) - \exp(-11.46y)$.

This equation holds on an interval $0 \leq x \leq 1$ for times $t \geq 0$. 
The PDE satisfies the initial conditions

\[ u_1(x, 0) = 1 \]
\[ u_2(x, 0) = 0 \]

and boundary conditions

\[ \frac{\partial u_1}{\partial x}(0, t) = 0 \]
\[ u_2(0, t) = 0 \]
\[ u_1(1, t) = 1 \]
\[ \frac{\partial u_2}{\partial x}(1, t) = 0 \]

In the form expected by pdepe, the equations are

\[
\begin{bmatrix}
1 \\
1
\end{bmatrix} \cdot \frac{\partial}{\partial t} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x} \begin{bmatrix}
0.024(\partial u_1/\partial x) \\
0.170(\partial u_2/\partial x)
\end{bmatrix} + \begin{bmatrix}
-F(u_1 - u_2) \\
F(u_1 - u_2)
\end{bmatrix}
\]

The boundary conditions on the partial derivatives of \( \mathbf{u} \) have to be written in terms of the flux. In the form expected by pdepe, the left boundary condition is

\[
\begin{bmatrix}
0 \\
u_2
\end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix}
0.024(\partial u_1/\partial x) \\
0.170(\partial u_2/\partial x)
\end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

and the right boundary condition is
The solution changes rapidly for small \( \varepsilon \). The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, the example must select the output times accordingly. There are boundary layers in the solution at both ends of \([0,1]\), so the example places mesh points near 0 and 1 to resolve these sharp changes. Often some experimentation is needed to select a mesh that reveals the behavior of the solution.

\[
\begin{bmatrix}
  u_1 - 1 \\
  0
\end{bmatrix} + \begin{bmatrix}
  0 \\
  1
\end{bmatrix} \cdot \begin{bmatrix}
  0.024 (\partial u_1 / \partial x) \\
  0.170 (\partial u_2 / \partial x)
\end{bmatrix} = \begin{bmatrix}
  0 \\
  0
\end{bmatrix}
\]

```matlab
function pdex4
    m = 0;
    x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
    t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

    sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
    u1 = sol(:,:,1);
    u2 = sol(:,:,2);

    figure
    surf(x,t,u1)
    title('u1(x,t)')
    xlabel('Distance x')
    ylabel('Time t')

    figure
    surf(x,t,u2)
    title('u2(x,t)')
    xlabel('Distance x')
    ylabel('Time t')
    
    function [c,f,s] = pdex4pde(x,t,u,DuDx)
    c = [1; 1];
    f = [0.024; 0.17] .* DuDx;
    y = u(1) - u(2);
```
\[ F = \exp(5.73y) - \exp(-11.47y); \]
\[ s = [-F; F]; \]

% --------------------------------------------------------------
function u0 = pdex4ic(x);
u0 = [1; 0];
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];

In this example, the PDEs, initial conditions, and boundary conditions are coded in subfunctions `pdex4pde`, `pdex4ic`, and `pdex4bc`.

The surface plots show the behavior of the solution components.
See Also
function_handle (@), pdeval, ode15s, odeset, odeget

References
pdeval

Purpose
Evaluate numerical solution of PDE using output of pdepe

Syntax
[uout, duoutdx] = pdeval(m, x, ui, xout)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Symmetry of the problem: slab = 0, cylindrical = 1, spherical = 2. This is the first input argument used in the call to pdepe.</td>
</tr>
<tr>
<td>xmesh</td>
<td>A vector [x0, x1, ..., xn] specifying the points at which the elements of ui were computed. This is the same vector with which pdepe was called.</td>
</tr>
<tr>
<td>ui</td>
<td>A vector sol(j,:,i) that approximates component i of the solution at time tf and mesh points xmesh, where sol is the solution returned by pdepe.</td>
</tr>
<tr>
<td>xout</td>
<td>A vector of points from the interval [x0,xn] at which the interpolated solution is requested.</td>
</tr>
</tbody>
</table>

Description
[uout, duoutdx] = pdeval(m, x, ui, xout) approximates the solution \( u_1 \) and its partial derivative \( \frac{\partial u_1}{\partial x} \) at points from the interval [x0,xn]. The pdeval function returns the computed values in uout and duoutdx, respectively.

Note
pdeval evaluates the partial derivative \( \frac{\partial u_1}{\partial x} \) rather than the flux \( f \). Although the flux is continuous, the partial derivative may have a jump at a material interface.

See Also
pdepe
Purpose

Example function of two variables

Syntax

$Z = \text{peaks;}$
$Z = \text{peaks}(n);$  
$Z = \text{peaks}(V);$  
$Z = \text{peaks}(X,Y);$  
$X,Y,Z] = \text{peaks;}$
$[X,Y,Z] = \text{peaks}(n);$  
$[X,Y,Z] = \text{peaks}(V);$  

Description

peaks is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating mesh, surf, pcolor, contour, and so on.

$Z = \text{peaks;}$ returns a 49-by-49 matrix.

$Z = \text{peaks}(n);$ returns an n-by-n matrix.

$Z = \text{peaks}(V);$ returns an n-by-n matrix, where $n = \text{length}(V)$.

$Z = \text{peaks}(X,Y);$ evaluates peaks at the given $X$ and $Y$ (which must be the same size) and returns a matrix the same size.
peaks(...) (with no output argument) plots the peaks function with surf.

[X,Y,Z] = peaks(...); returns two additional matrices, X and Y, for parametric plots, for example, surf(X,Y,Z,del2(Z)). If not given as input, the underlying matrices X and Y are

[X,Y] = meshgrid(V,V)

where V is a given vector, or V is a vector of length n with elements equally spaced from -3 to 3. If no input argument is given, the default n is 49.

See Also
meshgrid, surf
Purpose  Call Perl script using appropriate operating system executable

Syntax
perl('perlfile')
perl('perlfile',arg1,arg2,...)
result = perl(...)
[result, status] = perl(...)

Description  perl('perlfile') calls the Perl script perlfile, using the appropriate operating system Perl executable. Perl is included with the MATLAB software on Microsoft Windows systems, and thus MATLAB users
can run M-files containing the perl function. On UNIX systems, MATLAB calls the Perl interpreter available with the operating system.

perl('perlfile',arg1,arg2,...) calls the Perl script perlfile, using the appropriate operating system Perl executable, and passes the arguments arg1, arg2, and so on, to perlfile.

result = perl(...) returns the results of attempted Perl call to result.

[result, status] = perl(...) returns the results of attempted Perl call to result and its exit status to status.

It is sometimes beneficial to use Perl scripts instead of MATLAB code. The perl function allows you to run those scripts from MATLAB. Specific examples where you might choose to use a Perl script include:

- Perl script already exists
- Perl script preprocesses data quickly, formatting it in a way more easily read by MATLAB
- Perl has features not supported by MATLAB

### Examples

Given the Perl script, hello.pl:

```perl
$input = $ARGV[0];
print "Hello $input."
```

At the MATLAB command line, type:

```matlab
perl('hello.pl','World')
```

MATLAB displays:

```matlab
ans =
Hello World.
```

---

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

! (exclamation point), dos, regexp, system, unix
### Purpose
All possible permutations

### Syntax

\[ P = \text{perms}(v) \]

### Description

\( P = \text{perms}(v) \), where \( v \) is a row vector of length \( n \), creates a matrix whose rows consist of all possible permutations of the \( n \) elements of \( v \). Matrix \( P \) contains \( n! \) rows and \( n \) columns.

### Examples

The command \( \text{perms}(2:2:6) \) returns *all* the permutations of the numbers 2, 4, and 6:

\[
\begin{array}{ccc}
6 & 4 & 2 \\
6 & 2 & 4 \\
4 & 6 & 2 \\
4 & 2 & 6 \\
2 & 4 & 6 \\
2 & 6 & 4 \\
\end{array}
\]

### Limitations

This function is only practical for situations where \( n \) is less than about 15.

### See Also

nchoosek, permute, randperm

2-2784
Purpose
Rearrange dimensions of N-D array

Syntax
B = permute(A,order)

Description
B = permute(A,order) rearranges the dimensions of A so that they are in the order specified by the vector order. B has the same values of A but the order of the subscripts needed to access any particular element is rearranged as specified by order. All the elements of order must be unique.

Remarks
permute and ipermute are a generalization of transpose ( .' ) for multidimensional arrays.

Examples
Given any matrix A, the statement

permute(A,[2 1])

is the same as A.'.

For example:

A = [1 2; 3 4]; permute(A,[2 1])
ans =
     1     3
     2     4

The following code permutes a three-dimensional array:

X = rand(12,13,14);
Y = permute(X,[2 3 1]);
size(Y)
ans =
     13    14    12

See Also
ipermute, circshift, shiftdim, reshape
**Purpose**
Define persistent variable

**Syntax**
persistent X Y Z

**Description**
persistent X Y Z defines X, Y, and Z as variables that are local to the function in which they are declared; yet their values are retained in memory between calls to the function. Persistent variables are similar to global variables because the MATLAB software creates permanent storage for both. They differ from global variables in that persistent variables are known only to the function in which they are declared. This prevents persistent variables from being changed by other functions or from the MATLAB command line.

Persistent variables are cleared when the M-file is cleared from memory or when the M-file is changed. To keep an M-file in memory until MATLAB quits, use mlock.

If the persistent variable does not exist the first time you issue the persistent statement, it is initialized to the empty matrix.

It is an error to declare a variable persistent if a variable with the same name exists in the current workspace. MATLAB also errors if you declare any of a function’s input or output arguments as persistent within that same function. For example, the following persistent declaration is invalid:

```
function myfun(argA, argB, argC)
persistent argB
```

**Remarks**
There is no function form of the persistent command (i.e., you cannot use parentheses and quote the variable names).

**Example**
This function prompts a user to enter a directory name to use in locating one or more files. If the user has already entered this information, and it requires no modification, they do not need to enter it again. This is because the function stores it in a persistent variable (lastDir), and offers it as the default selection. Here is the function definition:
function find_file(file)
persistent lastDir

if isempty(lastDir)
    prompt = 'Enter directory: ';
else
    prompt = ['Enter directory[ ' lastDir ']: '];
end
response = input(prompt, 's');

if ~isempty(response)
    dirName = response;
else
    dirName = lastDir;
end

dir(strcat(dirName, file))
lastDir = dirName;

Execute the function twice. The first time, it prompts you to enter the information and does not offer a default:

    cd(matlabroot)

    find_file('is*.m')
Enter directory: toolbox/matlab/strfun/

    iscellstr.m ischar.m isletter.m isspace.m isstr.m isstrprop.m

The second time, it does offer a default taken from the persistent variable dirName:

    find_file('is*.m')
Enter directory[toolbox/matlab/strfun/]:
    toolbox/matlab/elmat/
**persistent**

isequal.m  isequalwithequalnans.m  isnan.m
isequal.m  isinf.m  isvector.m
isequal.m  isfinite.m  isscalar.m

**See Also**

global, clear, mislocked, mlock, munlock, isempty
Purpose
Ratio of circle’s circumference to its diameter, π

Syntax
pi

Description
pi returns the floating-point number nearest the value of π. The expressions 4*atan(1) and imag(log(-1)) provide the same value.

Examples
The expression sin(pi) is not exactly zero because pi is not exactly π.

    sin(pi)

    ans =

    1.2246e-16

See Also
ans, eps, i, Inf, j, NaN
**Purpose**
Pie chart

**GUI Alternatives**
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

```matlab
pie(X)
pie(X,explode)
pie(...,labels)
pie(axes_handle,...)
h = pie(...)
```

**Description**

`pie(X)` draws a pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie(X,explode)` offsets a slice from the pie. `explode` is a vector or matrix of zeros and nonzeros that correspond to `X`. A nonzero value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`. For example,

```matlab
pie(1:3,{'Taxes','Expenses','Profit'})
```

`pie(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = pie(...)` returns a vector of handles to patch and text graphics objects.
Remarks

The values in X are normalized via \( X / \text{sum}(X) \) to determine the area of each slice of the pie. If \( \text{sum}(X) \leq 1 \), the values in X directly specify the area of the pie slices. MATLAB draws only a partial pie if \( \text{sum}(X) < 1 \).

Examples

Emphasize the second slice in the chart by setting its corresponding explode element to 1.

```matlab
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie(x, explode)
colormap jet
```

See Also

pie3
Purpose

3-D pie chart

GUI

Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

pie3(X)
pie3(X,explode)
pie3(...,labels)
pie3(axes_handle,...)
h = pie3(...)

Description

pie3(X) draws a three-dimensional pie chart using the data in X. Each element in X is represented as a slice in the pie chart.

pie3(X,explode) specifies whether to offset a slice from the center of the pie chart. X(i,j) is offset from the center of the pie chart if explode(i,j) is nonzero. explode must be the same size as X.

pie3(...,labels) specifies text labels for the slices. The number of labels must equal the number of elements in X. For example,

    pie3(1:3,{'Taxes','Expenses','Profit'})

pie3(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes (gca).

h = pie3(...) returns a vector of handles to patch, surface, and text graphics objects.
Remarks

The values in \( X \) are normalized via \( X / \text{sum}(X) \) to determine the area of each slice of the pie. If \( \text{sum}(X) \leq 1 \), the values in \( X \) directly specify the area of the pie slices. MATLAB draws only a partial pie if \( \text{sum}(X) < 1 \).

Examples

Offset a slice in the pie chart by setting the corresponding `explode` element to 1:

```matlab
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie3(x,explode)
colormap hsv
```

See Also

`pie`
**Purpose**
Moore-Penrose pseudoinverse of matrix

**Syntax**

- `B = pinv(A)`
- `B = pinv(A,tol)`

**Definition**
The Moore-Penrose pseudoinverse is a matrix $B$ of the same dimensions as $A'$ satisfying four conditions:

- $A*B*A = A$
- $B*A*B = B$
- $A*B$ is Hermitian
- $B*A$ is Hermitian

The computation is based on $\text{svd}(A)$ and any singular values less than $\text{tol}$ are treated as zero.

**Description**

- `B = pinv(A)` returns the Moore-Penrose pseudoinverse of $A$.
- `B = pinv(A,tol)` returns the Moore-Penrose pseudoinverse and overrides the default tolerance, $\max(\text{size}(A))*\text{norm}(A) * \text{eps}$.

**Examples**
If $A$ is square and not singular, then `pinv(A)` is an expensive way to compute `inv(A)`. If $A$ is not square, or is square and singular, then `inv(A)` does not exist. In these cases, `pinv(A)` has some of, but not all, the properties of `inv(A)`.

If $A$ has more rows than columns and is not of full rank, then the overdetermined least squares problem

$$\text{minimize } \text{norm}(A*x-b)$$

does not have a unique solution. Two of the infinitely many solutions are

- $x = \text{pinv}(A)*b$
- $y = A\backslash b$
These two are distinguished by the facts that $\text{norm}(x)$ is smaller than the norm of any other solution and that $y$ has the fewest possible nonzero components.

For example, the matrix generated by

\[
A = \text{magic}(8); \quad A = A(:,1:6)
\]

is an 8-by-6 matrix that happens to have $\text{rank}(A) = 3$.

\[
A = \\
\begin{bmatrix}
64 & 2 & 3 & 61 & 60 & 6 \\
9 & 55 & 54 & 12 & 13 & 51 \\
17 & 47 & 46 & 20 & 21 & 43 \\
40 & 26 & 27 & 37 & 36 & 30 \\
32 & 34 & 35 & 29 & 28 & 38 \\
41 & 23 & 22 & 44 & 45 & 19 \\
49 & 15 & 14 & 52 & 53 & 11 \\
8 & 58 & 59 & 5 & 4 & 62 \\
\end{bmatrix}
\]

The right-hand side is $b = 260*\text{ones}(8,1)$,

\[
b = \\
\begin{bmatrix}
260 \\
260 \\
260 \\
260 \\
260 \\
260 \\
260 \\
260 \\
\end{bmatrix}
\]

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to $A*x = b$ would be a vector of all 1’s. With only six columns, the equations are still consistent, so a solution exists, but it is not all 1’s. Since the matrix is rank deficient, there are infinitely many solutions. Two of them are

\[
x = \text{pinv}(A)*b
\]
which is

\[
x = \\
1.1538 \\
1.4615 \\
1.3846 \\
1.3846 \\
1.4615 \\
1.1538 \\
\]

and

\[
y = A \backslash b
\]

which produces this result.

```
Warning: Rank deficient, rank = 3  tol =  1.8829e-013.
y = \\
4.0000 \\
5.0000 \\
0 \\
0 \\
0 \\
-1.0000
```

Both of these are exact solutions in the sense that \( \|A^*x - b\| \) and \( \|A^*y - b\| \) are on the order of roundoff error. The solution \( x \) is special because

\[ \|x\| = 3.2817 \]

is smaller than the norm of any other solution, including

\[ \|y\| = 6.4807 \]

On the other hand, the solution \( y \) is special because it has only three nonzero components.

**See Also**

inv, qr, rank, svd
Purpose

Givens plane rotation

Syntax

\[ [G,y] = \text{planerot}(x) \]

Description

\[ [G,y] = \text{planerot}(x) \] where \( x \) is a 2-component column vector, returns a 2-by-2 orthogonal matrix \( G \) so that \( y = G \cdot x \) has \( y(2) = 0 \).

Examples

\[
\begin{align*}
x &= [3 \ 4]; \\
[G,y] &= \text{planerot}(x')
\end{align*}
\]

\[
G =
\begin{bmatrix}
0.6000 & 0.8000 \\
-0.8000 & 0.6000
\end{bmatrix}
\]

\[
y =
\begin{bmatrix}
5 \\
0
\end{bmatrix}
\]

See Also

qrdelete, qrinsert
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Run M-file demo (deprecated; use <code>echodemo</code> instead)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>playshow filename</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>playshow filename</code> runs <code>filename</code>, which is a demo. Replace <code>playshow filename</code> with <code>echodemo filename</code>. Note that other arguments supported by <code>playshow</code> are not supported by <code>echodemo</code>.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td><code>demo</code>, <code>echodemo</code>, <code>helpbrowser</code></td>
</tr>
</tbody>
</table>
**Purpose**

2-D line plot

![Plot](plot.png)

**Contents**

“GUI Alternatives” on page 2-2799

“Description” on page 2-2800

“Backward-Compatiable Version” on page 2-2800

“Cycling Through Line Colors and Styles” on page 2-2801

“Prevent Resetting of Color and Styles with `hold all`” on page 2-2801

“Additional Information” on page 2-2802

“Specifying the Color and Size of Markers” on page 2-2802

“Specifying Tick-Mark Location and Labeling” on page 2-2803

“Adding Titles, Axis Labels, and Annotations” on page 2-2804

“See Also” on page 2-2805

**GUI Alternatives**

Use the Plot Selector `plot` to graph selected variables in the Workspace Browser and the Plot Catalog, accessed from the Figure Palette. Directly manipulate graphs in `plot edit` mode, and modify them using the Property Editor. For details, see Using Plot Edit Mode, and The Figure Palette in the MATLAB Graphics documentation, and also Creating Graphics from the Workspace Browser in the MATLAB Desktop documentation.

**Syntax**

```matlab
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineSpec,...)
plot(...,'PropertyName',PropertyValue,...)
plot(axes_handle,...)
```

```matlab
h = plot(...)
```
plot

hlines = plot('v6',...)

**Description**

`plot(Y)` plots the columns of `Y` versus their index if `Y` is a real number. If `Y` is complex, `plot(Y)` is equivalent to `plot(real(Y), imag(Y))`. In all other uses of `plot`, the imaginary component is ignored.

`plot(X1,Y1,...)` plots all lines defined by `Xn` versus `Yn` pairs. If only one of `Xn` or `Yn` is a matrix, the vector is plotted versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix. If `Xn` is a scalar and `Yn` is a vector, disconnected line objects are created and plotted as discrete points vertically at `Xn`.

`plot(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` is a line specification that determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs: `plot(X1,Y1,X2,Y2,LineSpec,X3,Y3).

**Note** See `LineSpec` for a list of line style, marker, and color specifiers.

```matlab
plot(...,'PropertyName',PropertyValue,...)```

sets properties to the specified property values for all lineseries graphics objects created by `plot`. (See the “Examples” on page 2-2802 section for examples.)

`plot(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = plot(...)` returns a column vector of handles to lineseries graphics objects, one handle per line.

**Backward-Compatible Version**

`hlines = plot('v6',...)` returns the handles to line objects instead of lineseries objects.
Note The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

Cycling Through Line Colors and Styles

If you do not specify a color when plotting more than one line, plot automatically cycles through the colors in the order specified by the current axes ColorOrder property. After cycling through all the colors defined by ColorOrder, plot then cycles through the line styles defined in the axesLineStyleOrder property.

The default LineStyleOrder property has a single entry (a solid line with no marker).

By default, MATLAB resets the ColorOrder and LineStyleOrder properties each time you call plot. If you want the changes you make to these properties to persist, you must define these changes as default values. For example,

```matlab
set(0,'DefaultAxesColorOrder',[0 0 0],...
     'DefaultAxesLineStyleOrder','-|-.|--|:');
```

sets the default ColorOrder to use only the color black and sets the LineStyleOrder to use solid, dash-dot, dash-dash, and dotted line styles.

Prevent Resetting of Color and Styles with hold all

The all option to the hold command prevents the ColorOrder and LineStyleOrder from being reset in subsequent plot commands. In the following sequence of commands, MATLAB continues to cycle through the colors defined by the axes ColorOrder property (see above).

```matlab
plot(rand(12,2))
hold all
```
plot(randn(12,2))

**Additional Information**

- See Creating Line Plots and Annotating Graphs for more information on plotting.
- See LineSpec for more information on specifying line styles and colors.

**Examples**

**Specifying the Color and Size of Markers**

You can also specify other line characteristics using graphics properties (see line for a description of these properties):

- **LineWidth** — Specifies the width (in points) of the line.
- **MarkerEdgeColor** — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- **MarkerFaceColor** — Specifies the color of the face of filled markers.
- **MarkerSize** — Specifies the size of the marker in units of points.

For example, these statements,

```matlab
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,...
     'MarkerEdgeColor','k',...
     'MarkerFaceColor','g',...
     'MarkerSize',10)
```

produce this graph.
**Specifying Tick-Mark Location and Labeling**

You can adjust the axis tick-mark locations and the labels appearing at each tick. For example, this plot of the sine function relabels the x-axis with more meaningful values:

\[
x = -\pi:.1:\pi;
\]
\[
y = \sin(x);
\]
\[
\text{plot}(x,y)
\]
\[
\text{set}(\text{gca},'\text{XTick}',-\pi:\pi/2:\pi)
\]
\[
\text{set}(\text{gca},'\text{XTickLabel}',\{-'\pi',-'\pi/2','0','\pi/2',\pi\})
\]

Now add axis labels and annotate the point -\pi/4, \sin(-\pi/4).
MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add an x- and y-axis label:

```matlab
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
text(-pi/4,sin(-pi/4),'
leftarrow sin(-\pi\div4)',...
'HorizontalAlignment','left')
```

Now change the line color to red by first finding the handle of the line object created by `plot` and then setting its `Color` property. In the same statement, set the `LineWidth` property to 2 points.

```matlab
set(findobj(gca,'Type','line','Color',[0 0 1]),...
    'Color','red',...
    'LineWidth',2)
```
See Also

axis, bar, grid, hold, legend, line, LineSpec, loglog, plot3, plotyy, semilogx, semilogy, subplot, title, xlabel, xlim, ylabel, ylim, zlabel, zlim, stem

See the text String property for a list of symbols and how to display them.

See the Plot Editor for information on plot annotation tools in the figure window toolbar.

See “Basic Plots and Graphs” on page 1-93 for related functions.
plot (timeseries)

**Purpose**
Plot time series

**Syntax**
- plot(ts)
- plot(tsc.tsname)
- plot(function)

**Description**
- `plot(ts)` plots the time-series data `ts` against time and interpolates values between samples by using either zero-order-hold ('zoh') or linear interpolation.
- `plot(tsc.tsname)` plots the timeseries object `tsname` that is part of the tscollection `tsc`.
- `plot(function)` accepts the modifiers used by the MATLAB plotting utility for numerical arrays. These modifiers can be specified as auxiliary inputs for modifying the appearance of the plot. See Examples below.

**Remarks**
Time-series events, when defined, are marked in the plot by a red circular marker. When you resize a timeseries plot to be narrower, the x-axis ticks and labels are readjusted so that they do not overlap one another. Unlike plots of regular variables, this behavior cannot be overridden (for example, by setting the axes `XTickMode` to 'Auto').

**Examples**
- `plot(ts,'-r*')` uses a regular line with the color red and marker '* to render the plot.
- `plot(ts,'ko','MarkerSize',3)` uses black circular markers of size 3 to render the plot.
Purpose

3-D line plot

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)

Description

The plot3 function displays a three-dimensional plot of a set of data points.

plot3(X1,Y1,Z1,...), where X1, Y1, Z1 are vectors or matrices, plots one or more lines in three-dimensional space through the points whose coordinates are the elements of X1, Y1, and Z1.

plot3(X1,Y1,Z1,LineSpec,...) creates and displays all lines defined by the Xn,Yn,Zn,LineSpec quads, where LineSpec is a line specification that determines line style, marker symbol, and color of the plotted lines.

plot3(...,'PropertyName',PropertyValue,...) sets properties to the specified property values for all line graphics objects created by plot3.

h = plot3(...) returns a column vector of handles to lineseries graphics objects, with one handle per object.
Remarks

If one or more of $X_1, Y_1, Z_1$ is a vector, the vectors are plotted versus the rows or columns of the matrix, depending whether the vectors’ lengths equal the number of rows or the number of columns.

You can mix $X_n, Y_n, Z_n$ triples with $X_n, Y_n, Z_n, LineSpec$ quads, for example,

```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

See `LineSpec` and `plot` for information on line types and markers.

Examples

Plot a three-dimensional helix.

```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
grid on
axis square
```
See Also

axis, bar3, grid, line, LineSpec, loglog, plot, semilogx, semilogy, subplot
plotbrowser

Purpose

Show or hide figure plot browser

GUI Alternatives

Click the larger **Plotting Tools** icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Open or close the **Plot Browser** tool from the figure’s **View** menu. For details, see “The Plot Browser” in the MATLAB Graphics documentation.

Syntax

```
plotbrowser('on')
plotbrowser('off')
plotbrowser('toggle')
plotbrowser
plotbrowser(figure_handle,...)
```

Description

- `plotbrowser('on')` displays the Plot Browser on the current figure.
- `plotbrowser('off')` hides the Plot Browser on the current figure.
- `plotbrowser('toggle')` or `plotbrowser` toggles the visibility of the Plot Browser on the current figure.
- `plotbrowser(figure_handle,...)` shows or hides the Plot Browser on the figure specified by `figure_handle`.

See Also

plottools, figurepalette, propertyeditor
**Purpose**

Interactively edit and annotate plots

**Syntax**

- `plotedit on`
- `plotedit off`
- `plotedit`
- `plotedit(h)`
- `plotedit('state')`
- `plotedit(h,'state')`

**Description**

`plotedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, change line styles, and add text, line, and arrow annotations.

`plotedit off` ends plot mode for the current figure.

`plotedit` toggles the plot edit mode for the current figure.

`plotedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plotedit('state')` specifies the `plotedit` state for the current figure. Values for `state` can be as shown.

<table>
<thead>
<tr>
<th>Value for state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on</code></td>
<td>Starts plot edit mode</td>
</tr>
<tr>
<td><code>off</code></td>
<td>Ends plot edit mode</td>
</tr>
<tr>
<td><code>showtoolsmenu</code></td>
<td>Displays the <strong>Tools</strong> menu in the menu bar</td>
</tr>
<tr>
<td><code>hidetoolsmenu</code></td>
<td>Removes the <strong>Tools</strong> menu from the menu bar</td>
</tr>
</tbody>
</table>

**Note** `hidetoolsmenu` is intended for GUI developers who do not want the **Tools** menu to appear in applications that use the figure window.
plotedit(h,'state') specifies the plotedit state for figure handle h.

**Remarks**

**Plot Editing Mode Graphical Interface Components**

To start plot edit mode, click this button.

Use the Edit, Insert, and Tools menus to add objects or edit existing objects in a graph.

Double-click on an object to select it.

Position labels, legends, and other objects by clicking and dragging.

Access object specific plot edit functions through context-sensitive pop-up menus.

**Examples**

Start plot edit mode for figure 2.

```
plotedit(2)
```

End plot edit mode for figure 2.

```
plotedit(2, 'off')
```
Hide the **Tools** menu for the current figure:

```matlab
plotedit('hidetoolsmenu')
```

**See Also**

`axes`, `line`, `open`, `plot`, `print`, `saveas`, `text`, `propedit`
Purpose

Scatter plot matrix

Syntax

plotmatrix(X,Y)
plotmatrix(X)
plotmatrix(...,'LineSpec')
[H,AX,BigAx,P] = plotmatrix(...)

Description

plotmatrix(X,Y) scatter plots the columns of X against the columns of Y. If X is p-by-m and Y is p-by-n, plotmatrix produces an n-by-m matrix of axes.

plotmatrix(X) is the same as plotmatrix(X,X), except that the diagonal is replaced by hist(X(:,i)).

plotmatrix(...,'LineSpec') uses a LineSpec to create the scatter plot. The default is 'r'.

[H,AX,BigAx,P] = plotmatrix(...) returns a matrix of handles to the objects created in H, a matrix of handles to the individual subaxes in AX, a handle to a big (invisible) axes that frames the subaxes in BigAx, and a matrix of handles for the histogram plots in P. BigAx is left as the current axes so that a subsequent title, xlabel, or ylabel command is centered with respect to the matrix of axes.

Examples

Generate plots of random data.

```matlab
x = randn(50,3); y = x*[-1 2 1;2 0 1;1 -2 3]';
plotmatrix(y,'*r')
```
See Also
scatter, scatter3
Plot Tools

Purpose
Show or hide plot tools

GUI
Click the larger Plotting Tools icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Individually select the Figure Palette, Plot Browser, and Property Editor tools from the figure’s View menu. For details, see “Plotting Tools — Interactive Plotting” in the MATLAB Graphics documentation.

Syntax
plottools('on')
plottools('off')
plottools
plottools(figure_handle,...)
plottools(...,'tool')

Description
plottools('on') displays the Figure Palette, Plot Browser, and Property Editor on the current figure, configured as you last used them.

plottools('off') hides the Figure Palette, Plot Browser, and Property Editor on the current figure.

plottools with no arguments, is the same as plottools('on')

plottools(figure_handle,...) displays or hides the plot tools on the specified figure instead of on the current figure.
plottools(...,'tool') operates on the specified tool only. *tool* can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

**Note** The first time you open the plotting tools, all three of them appear, grouped around the current figure as shown above. If you close, move, or undock any of the tools, MATLAB remembers the configuration you left them in and restores it when you invoke the tools for subsequent figures, both within and across MATLAB sessions.

**See Also** `figurepalette`, `plotbrowser`, `propertyeditor`
**Purpose**

2-D line plots with y-axes on both left and right side

**GUI Alternatives**

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see “Plotting Tools — Interactive Plotting” in the MATLAB Graphics documentation and “Creating Plots from the Workspace Browser” in the MATLAB Desktop Tools documentation.

**Syntax**

```matlab
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,function)
plotyy(X1,Y1,X2,Y2,'function1','function2')
[AX,H1,H2] = plotyy(...)```

**Description**

`plotyy(X1,Y1,X2,Y2)` plots `X1` versus `Y1` with y-axis labeling on the left and plots `X2` versus `Y2` with y-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,function)` uses the specified plotting function to produce the graph.

`function` can be either a function handle or a string specifying `plot`, `semilogx`, `semilogy`, `loglog`, `stem`, or any MATLAB function that accepts the syntax

```
h = function(x,y)
```

For example,

```matlab
plotyy(x1,y1,x2,y2,@loglog) % function handle
plotyy(x1,y1,x2,y2,'loglog') % string```

Function handles enable you to access user-defined subfunctions and can provide other advantages. See `@` for more information on using function handles.
plotyy(X1,Y1,X2,Y2,'function1','function2') uses
function1(X1,Y1) to plot the data for the left axis and
function2(X2,Y2) to plot the data for the right axis.

[AX,H1,H2] = plotyy(...) returns the handles of the two axes
created in AX and the handles of the graphics objects from each plot in
H1 and H2. AX(1) is the left axes and AX(2) is the right axes.

Examples

This example graphs two mathematical functions using plot as the
plotting function. The two y-axes enable you to display both sets of data
on one graph even though relative values of the data are quite different.

```matlab
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
[AX,H1,H2] = plotyy(x,y1,x,y2,'plot');
```

You can use the handles returned by plotyy to label the axes and set
the line styles used for plotting. With the axes handles you can specify
the YLabel properties of the left- and right-side y-axis:

```matlab
set(get(AX(1),'Ylabel'),'String','Slow Decay')
set(get(AX(2),'Ylabel'),'String','Fast Decay')
```

Use the xlabel and title commands to label the x-axis and add a title:

```matlab
xlabel('Time (\musec)')
title('Multiple Decay Rates')
```

Use the line handles to set the LineStyle properties of the left- and
right-side plots:

```matlab
set(H1,'LineStyle','--')
set(H2,'LineStyle',':')
```
See Also

plot, loglog, semilogx, semilogy, XAxisLocation, YAxisLocation

See “Using Multiple X- and Y-Axes” for more information.
Purpose
Simplex containing specified location

Syntax
SI = pointLocation(DT,QX)
SI = pointLocation(DT,QX,QY)
SI = pointLocation(DT,QX,QY,QZ)
[SI, BC] = pointLocation(DT,...)

Description
SI = pointLocation(DT,QX) returns the indices SI of the enclosing simplex (triangle/tetrahedron) for each query point location in QX. The enclosing simplex for point QX(k,:) is SI(k). pointLocation returns NaN for all points outside the convex hull.

SI = pointLocation(DT,QX,QY) and SI = pointLocation(DT,QX,QY,QZ) allow the query point locations to be specified in alternative column vector format when working in 2-D and 3-D.

[SI, BC] = pointLocation(DT,...) returns the barycentric coordinates BC.

Inputs
DT
Delaunay triangulation.
QX
Matrix of size mpts-by-ndim, mpts being the number of query points.

Outputs
SI
Column vector of length mpts containing the indices of the enclosing simplex for each query point. mpts is the number of query points.

BC
BC is a mpts-by-ndim matrix, each row BC(i,:) represents the barycentric coordinates of QX(i,:) with respect to the enclosing simplex SI(i).
Examples

Example 1

Create a 2-D Delaunay triangulation:

\[
\begin{align*}
X &= \text{rand}(10, 2); \\
\text{dt} &= \text{DelaunayTri}(X);
\end{align*}
\]

Find the triangles that contain specified query points:

\[
\begin{align*}
\text{qrypts} &= [0.25, 0.25; 0.5, 0.5]; \\
\text{triids} &= \text{pointLocation}(	ext{dt}, \text{qrypts})
\end{align*}
\]

Example 2

Create a 3-D Delaunay triangulation:

\[
\begin{align*}
x &= \text{rand}(10, 1); \\
y &= \text{rand}(10, 1); \\
z &= \text{rand}(10, 1); \\
\text{dt} &= \text{DelaunayTri}(x, y, z);
\end{align*}
\]

Find the triangles that contain specified query points and evaluate the barycentric coordinates:

\[
\begin{align*}
\text{qrypts} &= [0.25, 0.25, 0.25; 0.5, 0.5, 0.5]; \\
[\text{tetids}, \text{bcs}] &= \text{pointLocation}(	ext{dt}, \text{qrypts})
\end{align*}
\]

See Also

nearestNeighbor
Purpose
Transform polar or cylindrical coordinates to Cartesian

Syntax
[X,Y] = pol2cart(THETA,RHO)
[X,Y,Z] = pol2cart(THETA,RHO,Z)

Description
[X,Y] = pol2cart(THETA,RHO) transforms the polar coordinate data stored in corresponding elements of THETA and RHO to two-dimensional Cartesian, or xy, coordinates. The arrays THETA and RHO must be the same size (or either can be scalar). The values in THETA must be in radians.

xyz, [X,Y,Z] = pol2cart(THETA,RHO,Z) transforms the cylindrical coordinate data stored in corresponding elements of THETA, RHO, and Z to three-dimensional Cartesian, or coordinates. The arrays THETA, RHO, and Z must be the same size (or any can be scalar). The values in THETA must be in radians.

Algorithm
The mapping from polar and cylindrical coordinates to Cartesian coordinates is:

Polar to Cartesian Mapping
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)

Cylindrical to Cartesian Mapping
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
See Also: cart2pol, cart2sph, sph2cart
Purpose

Polar coordinate plot

GUI

Alternatives
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axes_handle,...)
h = polar(...)

Description

The polar function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

polar(theta,rho) creates a polar coordinate plot of the angle theta versus the radius rho. theta is the angle from the x-axis to the radius vector specified in radians; rho is the length of the radius vector specified in dataspace units.

polar(theta,rho,LineSpec) LineSpec specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

polar(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes (gca).

h = polar(...) returns the handle of a line object in h.

Remarks

Negative r values reflect through the origin, rotating by pi (since (theta,r) transforms to (r*cos(theta), r*sin(theta))). If you want different behavior, you can manipulate r prior to plotting. For example, you can make r equal to max(0,r) or abs(r).
Examples

Create a simple polar plot using a dashed red line:

```matlab
t = 0:.01:2*pi;
polar(t,sin(2*t).*cos(2*t),'--r')
```

See Also

cart2pol, compass, LineSpec, plot, pol2cart, rose
Purpose  Polynomial with specified roots

Syntax  

\[ p = \text{poly}(A) \]
\[ p = \text{poly}(r) \]

Description  

\[ p = \text{poly}(A) \] where \( A \) is an \( n \)-by-\( n \) matrix returns an \( n+1 \) element row vector whose elements are the coefficients of the characteristic polynomial, \( \text{det}(sI - A) \). The coefficients are ordered in descending powers: if a vector \( c \) has \( n+1 \) components, the polynomial it represents is
\[ c_1s^n + \ldots + c_n s + c_{n+1} \]

\[ p = \text{poly}(r) \] where \( r \) is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of \( r \).

Remarks  Note the relationship of this command to

\[ r = \text{roots}(p) \]

which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector \( p \). For vectors, \( \text{roots} \) and \( \text{poly} \) are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples  MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

\[ A = \]

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 0
\end{pmatrix}
\]

is returned in a row vector by \( \text{poly} \):

\[ p = \text{poly}(A) \]

\[ p = \]
The roots of this polynomial (eigenvalues of matrix A) are returned in a column vector by roots:

\[ r = \text{roots}(p) \]

\[ r = \begin{array}{l} 12.1229 \\ -5.7345 \\ -0.3884 \end{array} \]

Algorithm

The algorithms employed for poly and roots illustrate an interesting aspect of the modern approach to eigenvalue computation. poly(A) generates the characteristic polynomial of A, and roots(poly(A)) finds the roots of that polynomial, which are the eigenvalues of A. But both poly and roots use eig, which is based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If A is an n-by-n matrix, poly(A) produces the coefficients c(1) through c(n+1), with c(1) = 1, in

\[ \text{det}(\lambda I - A) = c_1\lambda^n + ... + c_n\lambda + c_{n+1} \]

The algorithm is

\[ z = \text{eig}(A); \]
\[ c = \text{zeros}(n+1,1); \; c(1) = 1; \]
\[ \text{for } j = 1:n \]
\[ \quad c(2:j+1) = c(2:j+1) - z(j)*c(1:j); \]
\[ \text{end} \]

This recursion is easily derived by expanding the product.

\[ (\lambda - \lambda_1)(\lambda - \lambda_2)...(\lambda - \lambda_n) \]
It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of $A$. This is true even if the eigenvalues of $A$ are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.

**See Also**

`conv`, `polyval`, `residue`, `roots`
**Purpose**  
Area of polygon

**Syntax**  
\[ \text{A} = \text{polyarea}(X,Y) \]  
\[ \text{A} = \text{polyarea}(X,Y,\text{dim}) \]

**Description**  
\( \text{A} = \text{polyarea}(X,Y) \) returns the area of the polygon specified by the vertices in the vectors \( X \) and \( Y \).

If \( X \) and \( Y \) are matrices of the same size, then \( \text{polyarea} \) returns the area of polygons defined by the columns \( X \) and \( Y \).

If \( X \) and \( Y \) are multidimensional arrays, \( \text{polyarea} \) returns the area of the polygons in the first nonsingleton dimension of \( X \) and \( Y \).

\( \text{A} = \text{polyarea}(X,Y,\text{dim}) \) operates along the dimension specified by scalar \( \text{dim} \).

**Examples**  
\[
\begin{align*}
L &= \text{linspace}(0,2.*\text{pi},6); \\
xv &= \cos(L)'; \\
yv &= \sin(L)'; \\
xv &= [xv; xv(1)]; \\
yv &= [yv; yv(1)]; \\
A &= \text{polyarea}(xv,yv); \\
\text{plot}(xv,yv); \text{title}(['\text{Area} = ' \text{num2str}(A)]); \text{axis image}
\end{align*}
\]
See Also  
convhull, inpolygon, rectint
**Purpose**
Polynomial derivative

**Syntax**
- `k = polyder(p)`
- `k = polyder(a,b)`
- `[q,d] = polyder(b,a)`

**Description**
The `polyder` function calculates the derivative of polynomials, polynomial products, and polynomial quotients. The operands `a`, `b`, and `p` are vectors whose elements are the coefficients of a polynomial in descending powers.

- `k = polyder(p)` returns the derivative of the polynomial `p`.
- `k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`.
- `[q,d] = polyder(b,a)` returns the numerator `q` and denominator `d` of the derivative of the polynomial quotient `b/a`.

**Examples**
The derivative of the product

\[(3x^2 + 6x + 9)(x^2 + 2x)\]

is obtained with

```matlab
a = [3 6 9];
b = [1 2 0];
k = polyder(a,b)
k =
    12   36   42   18
```

This result represents the polynomial

\[12x^3 + 36x^2 + 42x + 18\]

**See Also**
`conv`, `deconv`
Purpose
Polynomial eigenvalue problem

Syntax
[X,e] = polyeig(A0,A1,...Ap)
e = polyeig(A0,A1,...Ap)
[X, e, s] = polyeig(A0,A1,...,AP)

Description
[X,e] = polyeig(A0,A1,...Ap) solves the polynomial eigenvalue problem of degree p

\[(A_0 + \lambda A_1 + \ldots + \lambda^P A_p)x = 0\]

where polynomial degree p is a non-negative integer, and A0,A1,...,Ap are input matrices of order n. The output consists of a matrix X of size n-by-n*p whose columns are the eigenvectors, and a vector e of length n*p containing the eigenvalues.

If lambda is the jth eigenvalue in e, and x is the jth column of eigenvectors in X, then \((A_0 + \text{lambda}*A_1 + \ldots + \text{lambda}^p A_p)*x\) is approximately 0.

e = polyeig(A0,A1,...,Ap) is a vector of length n*p whose elements are the eigenvalues of the polynomial eigenvalue problem.

[X, e, s] = polyeig(A0,A1,...,AP) also returns a vector s of length p*n containing condition numbers for the eigenvalues. At least one of A0 and AP must be nonsingular. Large condition numbers imply that the problem is close to a problem with multiple eigenvalues.

Remarks
Based on the values of p and n, polyeig handles several special cases:

- p = 0, or polyeig(A) is the standard eigenvalue problem: eig(A).
- p = 1, or polyeig(A,B) is the generalized eigenvalue problem: eig(A,-B).
- n = 1, or polyeig(a0,a1,...,ap) for scalars a0, a1 ..., ap is the standard polynomial problem: roots([ap ... a1 a0]).
If both $A_0$ and $A_p$ are singular the problem is potentially ill-posed. Theoretically, the solutions might not exist or might not be unique. Computationally, the computed solutions might be inaccurate. If one, but not both, of $A_0$ and $A_p$ is singular, the problem is well posed, but some of the eigenvalues might be zero or infinite.

Note that scaling $A_0, A_1, \ldots, A_p$ to have $\text{norm}(A_i)$ roughly equal 1 may increase the accuracy of polyeig. In general, however, this cannot be achieved. (See Tisseur [3] for more detail.)

**Algorithm**

The polyeig function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of eig and qz for more on this.

**See Also**
condeig, eig, qz

**References**


Purpose
Polynomial curve fitting

Syntax
\[
p = \text{polyfit}(x,y,n)
\]
\[
[p,S] = \text{polyfit}(x,y,n)
\]
\[
[p,S,mu] = \text{polyfit}(x,y,n)
\]

Description
\[
p = \text{polyfit}(x,y,n)
\]
finds the coefficients of a polynomial \( p(x) \) of degree \( n \) that fits the data, \( p(x(i)) \) to \( y(i) \), in a least squares sense. The result \( p \) is a row vector of length \( n+1 \) containing the polynomial coefficients in descending powers

\[
p(x) = p_1x^n + p_2x^{n-1} + \ldots + p_n x + p_{n+1}
\]

\[
[p,S] = \text{polyfit}(x,y,n)
\]
returns the polynomial coefficients \( p \) and a structure \( S \) for use with \text{polyval} to obtain error estimates or predictions. Structure \( S \) contains fields \( \text{R}, \text{df}, \) and \( \text{normr} \), for the triangular factor from a QR decomposition of the Vandermonde matrix of \( X \), the degrees of freedom, and the norm of the residuals, respectively. If the data \( Y \) are random, an estimate of the covariance matrix of \( P \) is \( (\text{Rinv} \ast \text{Rinv'}) \ast \text{normr}^2/\text{df} \), where \( \text{Rinv} \) is the inverse of \( R \). If the errors in the data \( y \) are independent normal with constant variance, \text{polyval} produces error bounds that contain at least 50% of the predictions.

\[
[p,S,mu] = \text{polyfit}(x,y,n)
\]
finds the coefficients of a polynomial in

\[
\hat{x} = \frac{x - \mu_1}{\mu_2}
\]

where \( \mu_1 = \text{mean}(x) \) and \( \mu_2 = \text{std}(x) \). \( \mu \) is the two-element vector \([\mu_1, \mu_2] \). This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

Examples
This example involves fitting the error function, \( \text{erf}(x) \), by a polynomial in \( x \). This is a risky project because \( \text{erf}(x) \) is a bounded function, while polynomials are unbounded, so the fit might not be very good.
First generate a vector of \(x\) points, equally spaced in the interval \([0, 2.5]\); then evaluate \(\text{erf}(x)\) at those points.

\[
x = (0: 0.1: 2.5)';
y = \text{erf}(x);
\]

The coefficients in the approximating polynomial of degree 6 are

\[
p = \text{polyfit}(x,y,6)
\]

\[
p =
\begin{bmatrix}
0.0084 & -0.0983 & 0.4217 & -0.7435 & 0.1471 & 1.1064 & 0.0004
\end{bmatrix}
\]

There are seven coefficients and the polynomial is

\[
0.0084x^6 - 0.0983x^5 + 0.4217x^4 - 0.7435x^3 + 0.1471x^2 + 1.1064x + 0.0004
\]

To see how good the fit is, evaluate the polynomial at the data points with

\[
f = \text{polyval}(p,x);
\]

A table showing the data, fit, and error is

\[
table = [x \ y \ f \ y-f]
\]

\[
table =
\begin{bmatrix}
0 & 0 & 0.0004 & -0.0004 \\
0.1000 & 0.1125 & 0.1119 & 0.0006 \\
0.2000 & 0.2227 & 0.2223 & 0.0004 \\
0.3000 & 0.3286 & 0.3287 & -0.0001 \\
0.4000 & 0.4284 & 0.4288 & -0.0004 \\
... \\
2.1000 & 0.9970 & 0.9969 & 0.0001 \\
2.2000 & 0.9981 & 0.9982 & -0.0001 \\
2.3000 & 0.9989 & 0.9991 & -0.0003 \\
2.4000 & 0.9993 & 0.9995 & -0.0002
\end{bmatrix}
\]
So, on this interval, the fit is good to between three and four digits. Beyond this interval the graph shows that the polynomial behavior takes over and the approximation quickly deteriorates.

```matlab
x = (0: 0.1: 5)';
y = erf(x);
f = polyval(p,x);
plot(x,y,'o',x,f,'-')
axis([0 5 0 2])
```

### Algorithm

The `polyfit` M-file forms the Vandermonde matrix, $V$, whose elements are powers of $x$. 
\[ v_{i,j} = x_i^{n-j} \]

It then uses the backslash operator, \, to solve the least squares problem

\[ Vp \cong y \]

You can modify the M-file to use other functions of \( x \) as the basis functions.

**See Also**

poly, polyval, roots, lscov
**Purpose**
Integrate polynomial analytically

**Syntax**
\[
\text{polyint}(p,k) \\
\text{polyint}(p)
\]

**Description**
\text{polyint}(p,k) returns a polynomial representing the integral of polynomial p, using a scalar constant of integration k.
\text{polyint}(p) assumes a constant of integration k=0.

**See Also**
polyder, polyval, polyvalm, polyfit
**Purpose**
Polynomial evaluation

**Syntax**

\[
y = \text{polyval}(p,x) \\
y = \text{polyval}(p,x,[\text{}],\text{mu}) \\
[y,\text{delta}] = \text{polyval}(p,x,S) \\
[y,\text{delta}] = \text{polyval}(p,x,S,\text{mu})
\]

**Description**
y = polyval(p,x) returns the value of a polynomial of degree \(n\) evaluated at \(x\). The input argument \(p\) is a vector of length \(n+1\) whose elements are the coefficients in descending powers of the polynomial to be evaluated.

\[
y = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_n + 1
\]
x can be a matrix or a vector. In either case, polyval evaluates \(p\) at each element of \(x\).

\(y = \text{polyval}(p,x,[\text{}],\text{mu})\) uses \(\hat{x} = (x - \mu_1)/\mu_2\) in place of \(x\). In this equation, \(\mu_1 = \text{mean}(x)\) and \(\mu_2 = \text{std}(x)\). The centering and scaling parameters \(\text{mu} = [\mu_1, \mu_2]\) are optional output computed by polyfit.

\([y,\text{delta}] = \text{polyval}(p,x,S)\) and \([y,\text{delta}] = \text{polyval}(p,x,S,\text{mu})\) use the optional output structure \(S\) generated by polyfit to generate error estimates, \(y \pm \text{delta}\). If the errors in the data input to polyfit are independent normal with constant variance, \(y \pm \text{delta}\) contains at least 50% of the predictions.

**Remarks**
The polyvalm(p,x) function, with \(x\) a matrix, evaluates the polynomial in a matrix sense. See polyvalm for more information.

**Examples**
The polynomial \(p(x) = 3x^2 + 2x + 1\) is evaluated at \(x = 5, 7,\) and 9 with

\[
p = [3 \ 2 \ 1]; \\
polyval(p,[5 \ 7 \ 9])
\]

which results in
ans =

86   162   262

For another example, see polyfit.

See Also
polyfit, polyvalm
Purpose  Matrix polynomial evaluation

Syntax  \( Y = \text{polyvalm}(p,X) \)

Description  \( Y = \text{polyvalm}(p,X) \) evaluates a polynomial in a matrix sense. This is the same as substituting matrix \( X \) in the polynomial \( p \).

Polynomial \( p \) is a vector whose elements are the coefficients of a polynomial in descending powers, and \( X \) must be a square matrix.

Examples  The Pascal matrices are formed from Pascal's triangle of binomial coefficients. Here is the Pascal matrix of order 4.

\[
X = \text{pascal}(4)
\]
\[
X =
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 20 \\
\end{array}
\]

Its characteristic polynomial can be generated with the \( \text{poly} \) function.

\[
p = \text{poly}(X)
\]
\[
p =
\begin{array}{cccc}
1 & -29 & 72 & -29 & 1 \\
\end{array}
\]

This represents the polynomial \( x^4 - 29x^3 + 72x^2 - 29x + 1 \).

Pascal matrices have the curious property that the vector of coefficients of the characteristic polynomial is palindromic; it is the same forward and backward.

Evaluating this polynomial at each element is not very interesting.

\[
\text{polyval}(p,X)
\]
\[
\text{ans} =
\begin{array}{cccc}
16 & 16 & 16 & 16 \\
16 & 15 & -140 & -563 \\
16 & -140 & -2549 & -12089 \\
\end{array}
\]
But evaluating it in a matrix sense is interesting.

\[
polyvalm(p,X)
\]
\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array}
\]

The result is the zero matrix. This is an instance of the Cayley-Hamilton theorem: a matrix satisfies its own characteristic equation.

See Also

polyfit, polyval
Purpose
Base 2 power and scale floating-point numbers

Syntax
\[
X = \text{pow2}(Y) \\
X = \text{pow2}(F,E)
\]

Description
\(X = \text{pow2}(Y)\) returns an array \(X\) whose elements are 2 raised to the power \(Y\).

\(X = \text{pow2}(F,E)\) computes \(x = f \cdot 2^e\) for corresponding elements of \(F\) and \(E\). The result is computed quickly by simply adding \(E\) to the floating-point exponent of \(F\). Arguments \(F\) and \(E\) are real and integer arrays, respectively.

Remarks
This function corresponds to the ANSI C function ldexp() and the IEEE floating-point standard function scalbn().

Examples
For IEEE arithmetic, the statement \(X = \text{pow2}(F,E)\) yields the values:

<table>
<thead>
<tr>
<th>F</th>
<th>E</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pi/4</td>
<td>2</td>
<td>pi</td>
</tr>
<tr>
<td>-3/4</td>
<td>2</td>
<td>-3</td>
</tr>
<tr>
<td>1/2</td>
<td>-51</td>
<td>eps</td>
</tr>
<tr>
<td>1-eps/2</td>
<td>1024</td>
<td>realmax</td>
</tr>
<tr>
<td>1/2</td>
<td>-1021</td>
<td>realmin</td>
</tr>
</tbody>
</table>

See Also
log2, exp, hex2num, realmax, realmin
The arithmetic operators ^ and .^
Purpose
Array power

Syntax
Z = X.^Y

Description
Z = X.^Y denotes element-by-element powers. X and Y must have the same dimensions unless one is a scalar. A scalar is expanded to an array of the same size as the other input.

C = power(A,B) is called for the syntax 'A .^ B' when A or B is an object.

Note that for a negative value X and a non-integer value Y, if the abs(Y) is less than one, the power function returns the complex roots. To obtain the remaining real roots, use the nthroot function.

See Also
nthroot, realpow
**Purpose**
Evaluate piecewise polynomial

**Syntax**
\[ v = ppval(pp,xx) \]

**Description**
\( v = ppval(pp,xx) \) returns the value of the piecewise polynomial \( f \), contained in \( pp \), at the entries of \( xx \). You can construct \( pp \) using the functions \texttt{interp1}, \texttt{pchip}, \texttt{spline}, or the spline utility \texttt{mkpp}.

\( v \) is obtained by replacing each entry of \( xx \) by the value of \( f \) there. If \( f \) is scalar-valued, \( v \) is of the same size as \( xx \). \( xx \) may be N-dimensional.

If \( pp \) was constructed by \texttt{pchip}, \texttt{spline}, or \texttt{mkpp} using the orientation of non-scalar function values specified for those functions, then:

If \( f \) is \([D1, \ldots, Dr]\)-valued, and \( xx \) is a vector of length \( N \), then \( V \) has size \([D1, \ldots, Dr, N]\), with \( V(:, \ldots, :, J) \) the value of \( f \) at \( xx(J) \).

If \( f \) is \([D1, \ldots, Dr]\)-valued, and \( xx \) has size \([N1, \ldots, Ns]\), then \( V \) has size \([D1, \ldots, Dr, N1, \ldots, Ns]\), with \( V(:, \ldots, : , J1, \ldots, Js) \) the value of \( f \) at \( xx(J1, \ldots, Js) \).

If \( pp \) was constructed by \texttt{interp1} using the orientation of non-scalar function values specified for that function, then:

If \( f \) is \([D1, \ldots, Dr]\)-valued, and \( xx \) is a vector of length \( N \), then \( V \) has size \([N, D1, \ldots, Dr]\), with \( V(J, :, \ldots, :) \) the value of \( f \) at \( xx(J) \).

If \( f \) is \([D1, \ldots, Dr]\)-valued, and \( xx \) has size \([N1, \ldots, Ns]\), then \( V \) has size \([N1, \ldots, Ns, D1, \ldots, Dr]\), with \( V(J1, \ldots, Js, :, \ldots, :) \) the value of \( f \) at \( xx(J1, \ldots, Js) \).

**Examples**
Compare the results of integrating the function \( \cos \)

\[
\begin{align*}
a &= 0; \ b &= 10; \\
\text{int1} &= \text{quad}(@\text{cos}, a, b) \\
\text{int1} &= \ \\
&\quad -0.5440
\end{align*}
\]
with the results of integrating the piecewise polynomial pp that approximates the cosine function by interpolating the computed values x and y.

```matlab
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = quad(@(x)ppval(pp,x),a,b)
```

```
int2 =
-0.5485
```

`int1` provides the integral of the cosine function over the interval `[a, b]`, while `int2` provides the integral over the same interval of the piecewise polynomial pp.

**See Also**

`mkpp`, `spline`, `unmkpp`
**Purpose**

Directory containing preferences, history, and layout files

**Syntax**

```matlab
prefdir
d = prefdir
d = prefdir(1)
```

**Description**

`prefdir` returns the directory that contains

- Preferences for MATLAB and related products (`matlab.prf`)
- Command history file (`history.m`)
- MATLAB shortcuts (`shortcuts.xml`)
- MATLAB desktop layout files (`MATLABDesktop.xml` and `Your_Saved_LayoutMATLABLayout.xml`)
- Other related files

`d = prefdir` assigns to `d` the name of the directory containing preferences and related files.

`d = prefdir(1)` creates a directory for preferences and related files if one does not exist. If the directory does exist, the name is assigned to `d`.

**Remarks**

You must have write access to the preferences directory, or MATLAB generates an error in the Command Window when you try to change preferences.

The directory might be in a hidden folder, for example, `myname/.matlab/R2009a`. How to access hidden folders depends on your platform:

- On Microsoft Windows platforms, in any folder window, select **Tools > Folder Options**. Click the **View** tab, and under **Advanced** settings, select **Show hidden files and folders**. Then you should be able to see the folder returned by `prefdir`. 
On Apple Macintosh platforms, in the Finder, select **Go -> Go to Folder**. In the resulting dialog box, type the path returned by `prefdir` and press **Enter**.

The preferences directory MATLAB uses depends on the release. The preference directory naming and preference migration practice used from R13 through R14SP2 was changed starting in R14SP3 to address backwards compatibility problems. The differences are relevant primarily if you run multiple versions of MATLAB, and especially if one version is prior to R14SP3:

- For R2009a back through and including R2006a, and R14SP3, MATLAB uses the name of the release for the preference directory. For example, it uses `R2009a`, `R2008b`, ... through `R14SP3`. When you install R2009a, MATLAB migrates the files in the `R2008b` preferences directory to the `R2009a` preferences directory. While running R2009a through R14SP3, any changes made to files in those preferences directories (`R2009a` through `R14SP3`) are used only in their respective versions. As an example, commands you run in R2009a will *not* appear in the Command History when you run R2008b, and so on. The converse is also true.

Upon startup, MATLAB 7.8 (R2009a) looks for, and if found uses, the `R2009a` preferences directory. If not found, MATLAB creates an `R2009a` preferences directory. This happens when the `R2009a` preferences directory is deleted or does not exist for some other reason. MATLAB then looks for the `R2008b` preferences directory, and if found, migrates the `R2008b` preferences to the `R2009a` preferences. If it does not find the `R2008b` preferences directory, it uses the default preferences for R2009a. This process also applies when starting MATLAB 7.7 through 7.1.

If you want to use default preferences for R2009a, and do not want MATLAB to migrate preferences from R2008b, the `R2009a` preferences directory *must exist but be empty* when you start MATLAB. If you want to maintain some of your R2009a customizations, but restore the defaults for others, in the `R2009a` preferences directory, delete the files for which you want the defaults
to be restored. One file you might want to maintain is `history.m`—for more information about the file, see “Viewing Statements in the Command History Window” in the MATLAB Desktop Tools and Development Environment documentation.

- The R14 through R14SP2 releases all share the `R14` preferences directory. While running R14SP1, for example, any changes made to files in the preferences directory, `R14`, are used when you run R14SP2 and R14. As another example, commands you run in R14 appear in the Command History when you run R14SP2, and the converse is also true. The preferences are not used when you run R14SP3 or later versions because those versions each use their own preferences directories.

- All R13 releases use the `R13` preferences directory. While running R13SP1, for example, any changes made to files in the preferences directory, `R13`, are used when you run R13. As an example, commands you run in R13 will appear in the Command History when you run R13SP1, and the converse is true. The preferences are not used when you run any R14 or later releases because R14 and later releases use different preferences directories, and the converse is true.

**Examples**

Run

```
prefdir
```

MATLAB returns

```
ans =
C:\WINNT\Profiles\my_user_name\MATHWORKS\Application Data\MathWorks\MATLAB\R2009a
```

Running `dir` for the directory shows these files and others for MathWorks products:

```
.    history.m
..   matlab.prf
cwdhistory.m  MATLABDesktop.xml
shortcuts.xml  MATLAB EditorDesktop.xml
```
In MATLAB, run `cd(prefdir)` to change to that directory.

On Windows platforms, go directly to the preferences directory in Microsoft Windows Explorer by running `winopen(prefdir)`.

**See Also**

`preferences`, `winopen`

“Specifying Options for MATLAB Using Preferences” in the MATLAB Desktop Tools and Development Environment documentation
preferences

**Purpose**  
Open Preferences dialog box

**GUI Alternatives**  
As an alternative to the preferences function, select **File > Preferences** in the MATLAB desktop or any desktop tool.

**Syntax**  
preferences

**Description**  
preferences displays the Preferences dialog box, from which you can make changes to options for MATLAB and related products.

**See Also**  
prefdir

“Specifying Options for MATLAB Using Preferences” in the MATLAB Desktop Tools and Development Environment documentation
Purpose
Generate list of prime numbers

Syntax
p = primes(n)

Description
p = primes(n) returns a row vector of the prime numbers less than or equal to n. A prime number is one that has no factors other than 1 and itself.

Examples
p = primes(37)

p = 2 3 5 7 11 13 17 19 23 29 31 37

See Also
factor
**Purpose**

Print figure or save to file and configure printer defaults

**Contents**

“GUI Alternative” on page 2-2854

Syntax

“Description” on page 2-2854

“Printer Drivers” on page 2-2856

“Graphics Format Files” on page 2-2860

“Printing Options” on page 2-2864

“Paper Sizes” on page 2-2867

“Printing Tips” on page 2-2869

“Examples” on page 2-2871

“See Also” on page 2-2874

**GUI Alternative**

Select **File > Print** from the figure window to open the Print dialog box and **File > Print Preview** to open the Print Preview GUI. For details, see “How to Print or Export” in the MATLAB Graphics documentation.

**Syntax**

```plaintext
print
print filename
print -d(driver)
print -d(format)
print -d(format) filename
print -s(modelname)
print -options
print(...)  
[pcmd,dev] = printopt
```

**Description**

`print` and `printopt` produce hard-copy output. All arguments to the `print` command are optional. You can use them in any combination or order.
print sends the contents of the current figure, including bitmap representations of any user interface controls, to the printer using the device and system printing command defined by printopt.

print filename directs the output to the PostScript file designated by filename. If filename does not include an extension, print appends an appropriate extension.

print -d<driver> prints the figure using the specified printer driver, (such as color PostScript). If you omit -d<driver>, print uses the default value stored in printopt.m. The table in “Printer Drivers” on page 2-2856 lists all supported device types.

print -d<format> copies the figure to the system Clipboard (Microsoft Windows platforms only). To be valid, the format for this operation must be either -dmeta (Windows Enhanced Metafile) or -dbitmap (Windows Bitmap).

print -d<format> filename exports the figure to the specified file using the specified graphics format (such as TIFF). The table of “Graphics Format Files” on page 2-2860 lists all supported graphics file formats.

print -s<modelname> prints the current Simulink model <modelname>.

print -options specifies print options that modify the action of the print command. (For example, the -noui option suppresses printing of user interface controls.) “Printing Options” on page 2-2864 lists available options.

print(...) is the function form of print. It enables you to pass variables for any input arguments. This form is useful for passing file names and handles. See Batch Processing for an example.

[pcmd,dev] = printopt returns strings containing the current system-dependent printing command and output device. printopt is an M-file used by print to produce the hard-copy output. You can edit the M-file printopt.m to set your default printer type and destination.

pcmd and dev are platform-dependent strings. pcmd contains the command that print uses to send a file to the printer. dev contains the
printer driver or graphics format option for the print command. Their defaults are platform dependent.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Print Command</th>
<th>Driver or Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac and UNIX</td>
<td>lpr -r</td>
<td>-dps2</td>
</tr>
<tr>
<td>Windows</td>
<td>COPY /B %s LPT1:</td>
<td>-dwin</td>
</tr>
</tbody>
</table>

**Printer Drivers**

The following table shows the more widely used printer drivers supported by MATLAB software. If you do not specify a driver, the default setting shown in the previous table is used. For a list of all supported printer drivers, type `print -d` at the MATLAB prompt.

Some things to remember:

- As indicated in “Description” on page 2-2854 the `-d` switch specifies a printer driver or a graphics file format:
  - Specifying a printer driver without a file name or printer name (the `-P` option) sends the output formatted by the specified driver to your default printer, which may not be what you want to do.

**Note** On Windows systems, when you use the `-P` option to identify a printer to use, if you specify any driver other than `-dwin` or `-dwinc`, MATLAB writes the output to a file with an appropriate extension but does not send it to the printer. You can then copy that file to a printer.

- Specifying a `-dmeta` or a `-dbitmap` graphics format without a file name places the graphic on the system Clipboard, if possible (Windows platforms only).

- Specifying any other graphics format without a file name creates a file in the current directory with a name such as `figureN.fmt`,
where \( N \) is 1, 2, 3, ... and \( \text{fmt} \) indicates the format type, for example, \texttt{eps} or \texttt{png}.

- Several drivers come from a product called Ghostscript, which is shipped with MATLAB software. The last column indicates when Ghostscript is used.

- Not all drivers are supported on all platforms. Non support is noted in the first column of the table.

- If you specify a particular printer with the \texttt{-P} option and do not specify a driver, a default driver for that printer is selected, either by the operating system or by MATLAB, depending on the platform:
  - On MATLAB, the driver associated with this particular printing device is used.
  - On Macintosh and UNIX platforms, the driver specified in \texttt{printop.m} is used

See Selecting the Printer in the Graphics documentation for more information.

\textbf{Note} The MathWorks™ is planning to leverage existing operating system (OS) support for printer drivers and devices. As a result, the ability to specify certain print devices using the \texttt{print -d} command, and certain graphics formats using the \texttt{print -d} command and/or the \texttt{saveas} command, will be removed in a future release. In the following table, the affected formats have an asterisk (*) next to the \texttt{print} command option string. The asterisks provide a link to the Web site which supplies a form for users to give feedback about these changes.

<table>
<thead>
<tr>
<th>Printer Driver</th>
<th>Print Command Option String</th>
<th>Ghostscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon BubbleJet BJ10e</td>
<td>-dbj10e *</td>
<td>Yes</td>
</tr>
<tr>
<td>Printer Driver</td>
<td>Print Command Option String</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Canon BubbleJet BJ200 color</td>
<td>-dbj200 *</td>
<td>Yes</td>
</tr>
<tr>
<td>Canon Color BubbleJet BJC-70/BJC-600/BJC-4000</td>
<td>-dbjc600 *</td>
<td>Yes</td>
</tr>
<tr>
<td>Canon Color BubbleJet BJC-800</td>
<td>-dbjc800 *</td>
<td>Yes</td>
</tr>
<tr>
<td>Epson and compatible 9- or 24-pin dot matrix print drivers</td>
<td>-depson *</td>
<td>Yes</td>
</tr>
<tr>
<td>Epson and compatible 9-pin with interleaved lines (triple resolution)</td>
<td>-deps9high *</td>
<td>Yes</td>
</tr>
<tr>
<td>Epson LQ-2550 and compatible; color (not supported on HP-700)</td>
<td>-depsonc *</td>
<td>Yes</td>
</tr>
<tr>
<td>Fujitsu 3400/2400/1200</td>
<td>-depsonc *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DesignJet 650C color (not supported on Windows)</td>
<td>-ddnj650c *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet 500</td>
<td>-ddjet500 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet 500C (creates black and white output)</td>
<td>-dcdjmono *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet 500C (with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows)</td>
<td>-dcdjcolor *</td>
<td>Yes</td>
</tr>
<tr>
<td>Printer Driver</td>
<td>Print Command Option String</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>HP DeskJet 500C/540C color (not supported on Windows)</td>
<td>-dcdj500 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP Deskjet 550C color (not supported on Windows)</td>
<td>-dcdj550 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet and DeskJet Plus</td>
<td>-ddeskjet *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet</td>
<td>-dlaserjet *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet+</td>
<td>-dljetplus *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet IIIP</td>
<td>-dljet2p *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet III</td>
<td>-dljet3 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet 4, 5L and 5P</td>
<td>-dljet4 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet 5 and 6</td>
<td>-dpxlmono *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP PaintJet color</td>
<td>-dpaintjet *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP PaintJet XL color</td>
<td>-dpjx1 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP PaintJet XL color</td>
<td>-dpjetx1 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HP PaintJet XL300 color (not supported on Windows)</td>
<td>-dpjx1300 *</td>
<td>Yes</td>
</tr>
<tr>
<td>HPGL for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)</td>
<td>-dhpgl *</td>
<td>No</td>
</tr>
<tr>
<td>IBM® 9-pin Proprinter</td>
<td>-dibmpro *</td>
<td>Yes</td>
</tr>
<tr>
<td>Printer Driver</td>
<td>Print Command Option String</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>PostScript</strong> black and white</td>
<td>-dps</td>
<td>No</td>
</tr>
<tr>
<td><strong>PostScript</strong> color</td>
<td>-dpsc</td>
<td>No</td>
</tr>
<tr>
<td><strong>PostScript</strong> Level 2 black and white</td>
<td>-dps2</td>
<td>No</td>
</tr>
<tr>
<td><strong>PostScript</strong> Level 2 color</td>
<td>-dpsc2</td>
<td>No</td>
</tr>
<tr>
<td><strong>Windows color</strong> (Windows only)</td>
<td>-dwinc</td>
<td>No</td>
</tr>
<tr>
<td><strong>Windows monochrome</strong> (Windows only)</td>
<td>-dwin</td>
<td>No</td>
</tr>
</tbody>
</table>

**Tip** Generally, Level 2 PostScript files are smaller and are rendered more quickly when printing than Level 1 PostScript files. However, not all PostScript printers support Level 2, so determine the capabilities of your printer before using those drivers. Level 2 PostScript printing is the default for UNIX platforms. You can change this default by editing the printopt.m file. Likewise, if you want color PostScript printing to be the default instead of black-and-white PostScript printing, edit the line in the printopt.m file that reads `dev = '-dps2';` to be `dev = '-dpsc2';`.  

**Graphics Format Files**

To save your figure as a graphics format file, specify a format switch and file name. To set the resolution of the output file for a built-in MATLAB format, use the `-r` switch. (For example, `-r300` sets the output resolution to 300 dots per inch.) The `-r` switch is also supported for Windows Enhanced Metafiles, JPEG, TIFF and PNG files, but is not supported for Ghostscript raster formats. For more information, see “Printing and Exporting without a Display” on page 2-2863 and “Resolution Considerations” on page 2-2866.
Note When you print to a file, the file name must have fewer than 128 characters, including path name. When you print to a file in your current directory, the filename must have fewer than 126 characters, because MATLAB places './' or '.\' at the beginning of the filename when referring to it.

The following table shows the supported output formats for exporting from figures and the switch settings to use. In some cases, a format is available both as a MATLAB output filter and as a Ghostscript output filter. All formats except for EMF are supported on both PC and UNIX platforms.

<table>
<thead>
<tr>
<th>Graphics Format</th>
<th>Bitmap or Vector</th>
<th>Print Command Option String</th>
<th>MATLAB or Ghostscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMP monochrome BMP</td>
<td>Bitmap</td>
<td>-dbmpmono</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>BMP 24-bit BMP</td>
<td>Bitmap</td>
<td>-dbmp16m</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>BMP 8-bit (256-color) BMP (this format uses a fixed colormap)</td>
<td>Bitmap</td>
<td>-dbmp256</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>BMP 24-bit</td>
<td>Bitmap</td>
<td>-dbmp</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EMF</td>
<td>Vector</td>
<td>-dmeta</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS black and white</td>
<td>Vector</td>
<td>-deps</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS color</td>
<td>Vector</td>
<td>-depsc</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS Level 2 black and white</td>
<td>Vector</td>
<td>-deps2</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS Level 2 color</td>
<td>Vector</td>
<td>-depsc2</td>
<td>MATLAB</td>
</tr>
<tr>
<td>Graphics Format</td>
<td>Bitmap or Vector</td>
<td>Print Command Option String</td>
<td>MATLAB or Ghostscript</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>-----------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>HDF 24-bit</td>
<td>Bitmap</td>
<td>-dhdf</td>
<td>MATLAB</td>
</tr>
<tr>
<td>ILL (Adobe Illustrator)</td>
<td>Vector</td>
<td>-dill</td>
<td>MATLAB</td>
</tr>
<tr>
<td>JPEG 24-bit</td>
<td>Bitmap</td>
<td>-djpeg</td>
<td>MATLAB</td>
</tr>
<tr>
<td>PBM (plain format) 1-bit</td>
<td>Bitmap</td>
<td>-dpbm</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PBM (raw format) 1-bit</td>
<td>Bitmap</td>
<td>-dpbmraw</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX 1-bit</td>
<td>Bitmap</td>
<td>-dpcxmono</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX 24-bit color</td>
<td>Bitmap</td>
<td>-dpcx24b</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX file format, three 8-bit planes</td>
<td>Bitmap</td>
<td>-dpcx256</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX 8-bit newer color</td>
<td>Bitmap</td>
<td>-dpcx16</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX file format (256-color)</td>
<td>Bitmap</td>
<td>-dpcx16</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX Older color file format (EGA/VGA, 16-color)</td>
<td>Bitmap</td>
<td>-dpcx16</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PDF Color PDF file format</td>
<td>Vector</td>
<td>-dpdf</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PGM Portable Graymap (plain format)</td>
<td>Bitmap</td>
<td>-dpgm</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PGM Portable Graymap (raw format)</td>
<td>Bitmap</td>
<td>-dpgmraw</td>
<td>Ghostscript</td>
</tr>
</tbody>
</table>
The TIFF image format is supported on all platforms by almost all word processors for importing images. The \texttt{-dtiffn} variant writes an uncompressed TIFF. JPEG is a lossy, highly compressed format that is supported on all platforms for image processing and for inclusion into HTML documents on the Web. To create these formats, MATLAB renders the figure using the Z-buffer rendering method and the resulting bitmap is then saved to the specified file.

### Printing and Exporting without a Display

On a UNIX platform (including Macintosh), where you can start in MATLAB \texttt{nodisplay} mode (\texttt{matlab -nodisplay}), you can print using most of the drivers you can use with a display and export to most of the same file formats. The PostScript and Ghostscript devices all function in \texttt{nodisplay} mode on UNIX platforms. The graphic devices \texttt{-djpeg}, \texttt{-dpng}, \texttt{-dtiff} (compressed TIFF bitmaps), and \texttt{-tiff} (EPS with TIFF preview) work as well, but under \texttt{nodisplay} they use Ghostscript to generate output instead of using the drivers built into MATLAB. However, Ghostscript ignores the \texttt{-r} option when generating \texttt{-djpeg}, \texttt{-dpng}, \texttt{-dtiff}, and \texttt{-tiff} image files. This means that you cannot vary the resolution of image files when running in \texttt{nodisplay} mode.
Naturally, the Windows only -dwin and -dwinc output formats cannot be used on UNIX or Mac platforms with or without a display.

The same holds true on Windows platforms with the -noFigureWindows startup option. The -dwin, -dwinc, and -dsetup options operate as usual under -noFigureWindows. However, the printpreview GUI does not function in this mode.

The formats which you cannot generate in nodisplay mode on UNIX and Mac platforms are:

- bitmap (-dbitmap) — Windows bitmap file (except for Simulink models)
- bmp (-dbmp...) — Monochrome and color bitmaps
- hdf (-dhdf) — Hierarchical Data Format
- svg (-dsvg) — Scalable Vector Graphics file (except for Simulink models)
- tiffn (-dtiffn) — TIFF image file, no compression

In addition, uicontrols do not print or export in nodisplay mode.

### Printing Options

This table summarizes options that you can specify for print. The second column links to tutorials in “Printing and Exporting” in the MATLAB Graphics documentation that provide operational details. Also see “Resolution Considerations” on page 2-2866 for information on controlling output resolution.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-adobecset</td>
<td>PostScript devices only. Use PostScript default character set encoding. See “Early PostScript 1 Printers”.</td>
</tr>
<tr>
<td>-append</td>
<td>PostScript devices only. Append figure to existing PostScript file. See “Settings That Are Driver Specific”.</td>
</tr>
<tr>
<td><strong>Option</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td><code>-cmyk</code></td>
<td>PostScript devices only. Print with CMYK colors instead of RGB. See “Setting CMYK Color”.</td>
</tr>
<tr>
<td><code>-ddriver</code></td>
<td>Printing only. Printer driver to use. See “Printer Drivers” on page 2-2856 table.</td>
</tr>
<tr>
<td><code>-dformat</code></td>
<td>Exporting only. Graphics format to use. See “Graphics Format Files” table.</td>
</tr>
<tr>
<td><code>-dsetup</code></td>
<td>Windows printing only. Display the (platform-specific) Print Setup dialog. Settings you make in it are saved, but nothing is printed.</td>
</tr>
<tr>
<td><code>-fhandle</code></td>
<td>Handle of figure to print. Note that you cannot specify both this option and the <code>-swindowtitle</code> option. See “Which Figure Is Printed”.</td>
</tr>
<tr>
<td><code>-loose</code></td>
<td>PostScript and Ghostscript printing only. Use loose bounding box for PostScript output. See “Producing Uncropped Figures”.</td>
</tr>
<tr>
<td><code>-noui</code></td>
<td>Suppress printing of user interface controls. See “Excluding User Interface Controls”.</td>
</tr>
<tr>
<td><code>-opengl</code></td>
<td>Render using the OpenGL algorithm. Note that you cannot specify this method in conjunction with <code>-zbuffer</code> or <code>-painters</code>. See “Selecting a Renderer”.</td>
</tr>
<tr>
<td><code>-painters</code></td>
<td>Render using the Painter's algorithm. Note that you cannot specify this method in conjunction with <code>-zbuffer</code> or <code>-opengl</code>. See “Selecting a Renderer”.</td>
</tr>
<tr>
<td><code>-printer</code></td>
<td>Specify name of printer to use. See “Selecting the Printer”.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-rnumber</td>
<td>PostScript and built-in raster formats, and Ghostscript vector format only. Specify resolution in dots per inch. Defaults to 90 for Simulink, 150 for figures in image formats and when printing in Z-buffer or OpenGL mode, screen resolution for metafiles, and 864 otherwise. Use -r0 to specify screen resolution. For details, see “Resolution Considerations” on page 2-2866 and “Setting the Resolution”.</td>
</tr>
<tr>
<td>-swindowtitle</td>
<td>Specify name of Simulink system window to print. Note that you cannot specify both this option and the -fhandle option. See “Which Figure Is Printed”.</td>
</tr>
<tr>
<td>-v</td>
<td>Windows printing only. Display the Windows Print dialog box. The v stands for “verbose mode.”</td>
</tr>
<tr>
<td>-zbuffer</td>
<td>Render using the Z-buffer algorithm. Note that you cannot specify this method in conjunction with -opengl or -painters. See “Selecting a Renderer”.</td>
</tr>
</tbody>
</table>

**Resolution Considerations**

Use -rnumber to specify the resolution of the generated output. In general, using a higher value will yield higher quality output but at the cost of larger output files. It affects the resolution and output size of all MATLAB built-in raster formats (which are identified in column four of the table in “Graphics Format Files” on page 2-2860).
Note  Built-in graphics formats are generated directly from MATLAB without conversion through the Ghostscript library. Also, in headless (nodisplay) mode, writing to certain image formats is not done by built-in drivers, as it is when a display is being used. These formats are -djepg, -dtiff, and -dpng. Furthermore, the -dhdf and -dbmp formats cannot be generated in headless mode (but you can substitute -dbmp16m for -dbmp). See “Printing and Exporting without a Display” on page 2-2863 for details on printing when not using a display.

Unlike the built-in MATLAB formats, graphic output generated via Ghostscript does not directly obey -r option settings. However, the intermediate PostScript file generated by MATLAB as input for the Ghostscript processor is affected by the -r setting and thus can indirectly influence the quality of the final Ghostscript generated output.

The effect of the -r option on output quality can be subtle at ordinary magnification when using the OpenGL or ZBuffer renderers and writing to one of the MATLAB built-in raster formats, or when generating vector output that contains an embedded raster image (for example, PostScript or PDF). The effect of specifying higher resolution is more apparent when viewing the output at higher magnification or when printed, since a larger -r setting provides more data to use when scaling the image.

When generating fully vectorized output (as when using the Painters renderer to output a vector format such as PostScript or PDF), the resolution setting affects the degree of detail of the output; setting resolution higher generates crisper output (but small changes in the resolution may have no observable effect). For example, the gap widths of lines that do not use a solid (‘-’) linestyle can be affected.

Paper Sizes

MATLAB printing supports a number of standard paper sizes. You can select from the following list by setting the PaperType property of the figure or selecting a supported paper size from the Print dialog box.
<table>
<thead>
<tr>
<th>Property Value</th>
<th>Size (Width by Height)</th>
</tr>
</thead>
<tbody>
<tr>
<td>usletter</td>
<td>8.5 by 11 inches</td>
</tr>
<tr>
<td>uslegal</td>
<td>8.5 by 14 inches</td>
</tr>
<tr>
<td>tabloid</td>
<td>11 by 17 inches</td>
</tr>
<tr>
<td>A0</td>
<td>841 by 1189 mm</td>
</tr>
<tr>
<td>A1</td>
<td>594 by 841 mm</td>
</tr>
<tr>
<td>A2</td>
<td>420 by 594 mm</td>
</tr>
<tr>
<td>A3</td>
<td>297 by 420 mm</td>
</tr>
<tr>
<td>A4</td>
<td>210 by 297 mm</td>
</tr>
<tr>
<td>A5</td>
<td>148 by 210 mm</td>
</tr>
<tr>
<td>B0</td>
<td>1029 by 1456 mm</td>
</tr>
<tr>
<td>B1</td>
<td>728 by 1028 mm</td>
</tr>
<tr>
<td>B2</td>
<td>514 by 728 mm</td>
</tr>
<tr>
<td>B3</td>
<td>364 by 514 mm</td>
</tr>
<tr>
<td>B4</td>
<td>257 by 364 mm</td>
</tr>
<tr>
<td>B5</td>
<td>182 by 257 mm</td>
</tr>
<tr>
<td>arch-A</td>
<td>9 by 12 inches</td>
</tr>
<tr>
<td>arch-B</td>
<td>12 by 18 inches</td>
</tr>
<tr>
<td>arch-C</td>
<td>18 by 24 inches</td>
</tr>
<tr>
<td>arch-D</td>
<td>24 by 36 inches</td>
</tr>
<tr>
<td>arch-E</td>
<td>36 by 48 inches</td>
</tr>
<tr>
<td>A</td>
<td>8.5 by 11 inches</td>
</tr>
<tr>
<td>B</td>
<td>11 by 17 inches</td>
</tr>
<tr>
<td>C</td>
<td>17 by 22 inches</td>
</tr>
<tr>
<td>D</td>
<td>22 by 34 inches</td>
</tr>
<tr>
<td>E</td>
<td>34 by 43 inches</td>
</tr>
</tbody>
</table>
**Printing Tips**

**Figures with Resize Functions**

The print command produces a warning when you print a figure having a callback routine defined for the figure **ResizeFcn**. To avoid the warning, set the figure **PaperPositionMode** property to auto or select **Match Figure Screen Size** in the File > Page Setup dialog box.

**Troubleshooting Microsoft Windows Printing**

If you encounter problems such as segmentation violations, general protection faults, or application errors, or the output does not appear as you expect when using Microsoft printer drivers, try the following:

- If your printer is PostScript compatible, print with one of the MATLAB built-in PostScript drivers. There are various PostScript device options that you can use with `print`, which all start with `-dps`.
- The behavior you are experiencing might occur only with certain versions of the print driver. Contact the print driver vendor for information on how to obtain and install a different driver.
- Try printing with one of the MATLAB built-in Ghostscript devices. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet, PCX, Canon BubbleJet, and so on.
- Copy the figure as a Windows Enhanced Metafile using the **Edit > Copy Figure** menu item on the figure window menu or the `print -dmeta` option at the command line. You can then import the file into another application for printing.

You can set copy options in the figure’s **File > Preferences > Copying Options** dialog box. The Windows Enhanced Metafile Clipboard format produces a better quality image than Windows Bitmap.

**Printing MATLAB GUIs**

You can generally obtain better results when printing a figure window that contains MATLAB **uicontrols** by setting these key properties:
• Set the figure PaperPositionMode property to auto. This ensures that the printed version is the same size as the on-screen version. With PaperPositionMode set to auto MATLAB, does not resize the figure to fit the current value of the PaperPosition. This is particularly important if you have specified a figure ResizeFcn, because if MATLAB resizes the figure during the print operation, ResizeFcn is automatically called.

To set PaperPositionMode on the current figure, use the command:

```matlab
set(gcf,'PaperPositionMode','auto')
```

• Set the figure InvertHardcopy property to off. By default, MATLAB changes the figure background color of printed output to white, but does not change the color of uicontrols. If you have set the background color, for example, to match the gray of the GUI devices, you must set InvertHardcopy to off to preserve the color scheme.

To set InvertHardcopy on the current figure, use the command:

```matlab
set(gcf,'InvertHardcopy','off')
```

• Use a color device if you want lines and text that are in color on the screen to be written to the output file as colored objects. Black and white devices convert colored lines and text to black or white to provide the best contrast with the background and to avoid dithering.

• Use the print command’s -loose option to keep a bounding box from being too tightly wrapped around objects contained in the figure. This is important if you have intentionally used space between uicontrols or axes and the edge of the figure and you want to maintain this appearance in the printed output.

If you print or export in nodisplay mode, none of the uicontrols the figure has will be visible. If you run code that adds uicontrols to a figure when the figure is invisible, the controls will not print until the figure is made visible.
Printing Interpolated Shading with PostScript Drivers

You can print MATLAB surface objects (such as graphs created with \texttt{surf} or \texttt{mesh}) using interpolated colors. However, only patch objects that are composed of triangular faces can be printed using interpolated shading.

Printed output is always interpolated in RGB space, not in the colormap colors. This means that if you are using indexed color and interpolated face coloring, the printed output can look different from what is displayed on screen.

PostScript files generated for interpolated shading contain the color information of the graphics object’s vertices and require the printer to perform the interpolation calculations. This can take an excessive amount of time and in some cases, printers might timeout before finishing the print job. One solution to this problem is to interpolate the data and generate a greater number of faces, which can then be flat shaded.

To ensure that the printed output matches what you see on the screen, print using the \texttt{-zbuffer} option. To obtain higher resolution (for example, to make text look better), use the \texttt{-r} option to increase the resolution. There is, however, a tradeoff between the resolution and the size of the created PostScript file, which can be quite large at higher resolutions. The default resolution of 150 dpi generally produces good results. You can reduce the size of the output file by making the figure smaller before printing it and setting the figure \texttt{PaperPositionMode} to \texttt{auto}, or by just setting the \texttt{PaperPosition} property to a smaller size.

Examples

Specifying the Figure to Print

You can print a noncurrent figure by specifying the figure’s handle. If a figure has the title “Figure 2”, its handle is 2. The syntax is:

\begin{verbatim}
print -fhandle
\end{verbatim}

This example prints the figure whose handle is 2, regardless of which figure is the current figure:
print -f2

**Note** You must use the -f option if the figure’s handle is hidden (i.e., its HandleVisibility property is set to off).

This example saves the figure with the handle -f2 to a PostScript file named Figure2, which can be printed later:

```
print -f2 -dps 'Figure2.ps'
```

If the figure uses noninteger handles, use the `figure` command to get its value, and then pass it in as the first argument. For example:

```
h = figure('IntegerHandle','off')
print h -dps2
```

You can also pass a figure handle as a variable to the function form of `print`. For example:

```
h = figure; plot(1:4,5:8)
print(h)
```

This example uses the function form of `print` to enable a file name to be passed in as a variable:

```
filename = 'mydata';
print('-f3', '-dpssc', filename);
```

(Because a file name is specified, the figure will be printed to a file.)

**Specifying the Model to Print**

To print a noncurrent Simulink model, use the -s option with the title of the window. For example, this command prints the Simulink window titled f14:

```
print -sf14
```
If the window title includes any spaces, you must call the function form rather than the command form of `print`. For example, this command saves the Simulink window title `Thruster Control`:

```
print('-sThruster Control')
```

To print the current system, use:

```
print -s
```

For information about issues specific to printing Simulink windows, see the Simulink documentation.

**Printing Figures at Screen Size**

This example prints a surface plot with interpolated shading. Setting the current figure’s (`gcf`) `PaperPositionMode` to `auto` enables you to resize the figure window and print it at the size you see on the screen. See “Printing Options” on page 2-2864 and “Printing Interpolated Shading with PostScript Drivers” on page 2-2871 for information on the `-zbuffer` and `-r200` options.

```
surf(peaks)
shading interp
set(gcf,'PaperPositionMode','auto')
print -dpsc2 -zbuffer -r200
```

For additional details, see “Printing Images” in the MATLAB Graphics documentation.

**Batch Processing**

You can use the function form of `print` to pass variables containing file names. For example, this `for` loop uses file names stored in a cell array to create a series of graphs and prints each one with a different file name:

```
fnames = {'file1', 'file2', 'file3'};
for k=1:length(fnames)
```

surf(peaks)
print('-dtiff','-r200',fnames{k})
end

**Tiff Preview**

The command

```
print -depsc -tiff -r300 picture1
```

saves the current figure at 300 dpi, in a color Encapsulated PostScript file named `picture1.eps`. The `-tiff` option creates a 72 dpi TIFF preview, which many word processor applications can display on screen after you import the EPS file. This enables you to view the picture on screen within your word processor and print the document to a PostScript printer using a resolution of 300 dpi.

**See Also**

`figure`, `hgsave`, `imwrite`, `orient`, `printdlg`, `printopt`, `saveas`
Purpose
Print dialog box

Syntax
printdlg
printdlg(fig)
printdlg('-crossplatform', fig)
printdlg('-setup', fig)

Description
printdlg prints the current figure.
printdlg(fig) creates a modal dialog box from which you can print the figure window identified by the handle fig. Note that uimenu do not print.
printdlg('-crossplatform', fig) displays the standard cross-platform MATLAB printing dialog rather than the built-in printing dialog box for Microsoft Windows computers. Insert this option before the fig argument.
printdlg('-setup', fig) forces the printing dialog to appear in a setup mode. Here one can set the default printing options without actually printing.

Note
A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in the MATLAB Figure Properties.

See Also
pagesetupdlg, printpreview
Purpose
Preview figure to print

Contents
“GUI Alternative” on page 2-2876
“Description” on page 2-2876
“Right Pane Controls” on page 2-2877
“The Layout Tab” on page 2-2878
“The Lines/Text Tab” on page 2-2879
“The Color Tab” on page 2-2881
“The Advanced Tab” on page 2-2883
“See Also” on page 2-2884

GUI Alternative
Use File > Print Preview on the figure window menu to access the Print Preview dialog box, described below. For details, see “Using Print Preview” in the MATLAB Graphics documentation.

Syntax
printpreview
printpreview(f)

Description
printpreview displays a dialog box showing the figure in the currently active figure window as it will print. A scaled version of the figure displays in the right-hand pane of the GUI.

printpreview(f) displays a dialog box showing the figure having the handle f as it will print.

Use the Print Preview dialog box, shown below, to control the layout and appearance of figures before sending them to a printer or print file. Controls are grouped into four tabbed panes: Layout, Lines/Text, Color, and Advanced.
Right Pane Controls

You can position and scale plots on the printed page using the rulers in the right-hand pane of the Print Preview dialog. Use the outer ruler handlebars to change margins. Moving them changes plot proportions. Use the center ruler handlebars to change the position of the plot on the page. Plot proportions do not change, but you can move portions of
the plot off the paper. The buttons on that pane let you refresh the
plot, close the dialog (preserving all current settings), print the page
immediately, or obtain context-sensitive help. Use the Zoom box and
scroll bars to view and position page elements more precisely.

The Layout Tab

Use the Layout tab, shown above, to control the paper format and
placement of the plot on printed pages. The following table summarizes
the Layout options:

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placement</td>
<td><strong>Auto</strong></td>
<td>Let MATLAB decide placement of plot on page</td>
</tr>
<tr>
<td></td>
<td><strong>Use manual...</strong></td>
<td>Specify position parameters for plot on page</td>
</tr>
<tr>
<td></td>
<td><strong>Top, Left, Width, Height</strong></td>
<td>Standard position parameters in current units</td>
</tr>
<tr>
<td></td>
<td><strong>Use defaults</strong></td>
<td>Revert to default position</td>
</tr>
<tr>
<td></td>
<td><strong>Fill page</strong></td>
<td>Expand figure to fill printable area (see note below)</td>
</tr>
<tr>
<td></td>
<td><strong>Fix aspect ratio</strong></td>
<td>Correct height/width ratio</td>
</tr>
<tr>
<td></td>
<td><strong>Center</strong></td>
<td>Center plot on printed page</td>
</tr>
<tr>
<td>Paper</td>
<td><strong>Format</strong></td>
<td>U.S. and ISO® sheet size selector</td>
</tr>
<tr>
<td></td>
<td><strong>Width, Height</strong></td>
<td>Sheet size in current units</td>
</tr>
<tr>
<td>Units</td>
<td><strong>Inches</strong></td>
<td>Use inches as units for dimensions and positions</td>
</tr>
<tr>
<td></td>
<td><strong>Centimeters</strong></td>
<td>Use centimeters as units for dimensions and positions</td>
</tr>
<tr>
<td></td>
<td><strong>Points</strong></td>
<td>Use points as units for dimensions and positions</td>
</tr>
<tr>
<td>Orientation</td>
<td><strong>Portrait</strong></td>
<td>Upright paper orientation</td>
</tr>
<tr>
<td>Group</td>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Landscape</td>
<td>Sideways paper orientation</td>
</tr>
<tr>
<td></td>
<td>Rotated</td>
<td>Currently the same as Landscape</td>
</tr>
</tbody>
</table>

**Note**  Selecting the Fill page option changes the PaperPosition property to fill the page, allowing objects in normalized units to expand to fill the space. If an object within the figure has an absolute size, for example a table, it can overflow the page when objects with normalized units expand. To avoid having objects fall off the page, do not use Fill page under such circumstances.

**The Lines/Text Tab**

Use the Lines/Text tab, shown below, to control the line weights, font characteristics, and headers for printed pages. The following table summarizes the Lines/Text options:
### Group Option Description

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td><strong>Line Width</strong></td>
<td>Scale all lines by a percentage from 0 upward (100 being no change), print lines at a specified point size, or default line widths used on the plot</td>
</tr>
<tr>
<td></td>
<td><strong>Min Width</strong></td>
<td>Smallest line width (in points) to use when printing; defaults to 0.5 point</td>
</tr>
<tr>
<td>Text</td>
<td><strong>Font Name</strong></td>
<td>Select a system font for all text on plot, or default to fonts currently used on the plot</td>
</tr>
<tr>
<td>Group</td>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Font Size</td>
<td>Scale all text by a percentage from 0 upward (100 being no change), print text at a specified point size, or default to this used on the plot.</td>
<td></td>
</tr>
<tr>
<td>Font Weight</td>
<td>Select Normal ... Bold font styling for all text from drop-down menu or default to the font weights used on the plot.</td>
<td></td>
</tr>
<tr>
<td>Font Angle</td>
<td>Select Normal, Italic or Oblique font styling for all text from drop-down menu or default to the font angles used on the plot.</td>
<td></td>
</tr>
<tr>
<td>Header</td>
<td>Header Text</td>
<td>Type the text to appear on the header at the upper left of printed pages, or leave blank for no header.</td>
</tr>
<tr>
<td></td>
<td>Date Style</td>
<td>Select a date format to have today’s date appear at the upper left of printed pages, or none for no date.</td>
</tr>
</tbody>
</table>

**The Color Tab**

Use the Color tab, shown below, to control how colors are printed for lines and backgrounds. The following table summarizes the Color options:
<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color Scale</td>
<td><strong>Black and White</strong></td>
<td>Select to print lines and text in black and white, but use color for patches and other objects</td>
</tr>
<tr>
<td></td>
<td><strong>Gray Scale</strong></td>
<td>Convert colors to shades of gray on printed pages</td>
</tr>
</tbody>
</table>
### Group Option Description

- **Color**
  - Print everything in color, matching colors on plot; select RGB (default) or CMYK color model for printing.

- **Background Color**
  - **Same as figure**
    - Print the figure’s background color as it is.
  - **Custom**
    - Select a color name, or type a colorspec for the background; white (default) implies no background color, even on colored paper.

---

### The Advanced Tab

Use the **Advanced** tab, shown below, to control finer details of printing, such as limits and ticks, renderer, resolution, and the printing of UIControls. The following table summarizes the **Advanced** options:

- **Axes limits and ticks**
  - **Recompute limits and ticks**
  - **Keep screen limits and ticks**

- **Miscellaneous**
  - **Renderer**
    - **auto**
  - **Resolution**
    - **auto**
  - **Print UIControls**

<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes limits and ticks</td>
<td>Recompute limits and ticks</td>
<td>Redraw x- and y-axes ticks and limits based on printed plot size (default)</td>
</tr>
</tbody>
</table>
### See Also

printdlg, pagesetupdlg

For more information, see How to Print or Export in the MATLAB Graphics documentation.
**Purpose**  
Product of array elements

**Syntax**  
B = prod(A)  
B = prod(A,dim)

**Description**  
B = prod(A) returns the products along different dimensions of an array.

If A is a vector, prod(A) returns the product of the elements.

If A is a matrix, prod(A) treats the columns of A as vectors, returning a row vector of the products of each column.

If A is a multidimensional array, prod(A) treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

B = prod(A,dim) takes the products along the dimension of A specified by scalar dim.

**Examples**  
The magic square of order 3 is

M = magic(3)

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

The product of the elements in each column is

```
prod(M) =
   96  45  84
```

The product of the elements in each row can be obtained by:

```
prod(M,2) =
   48
```
See Also

cumprod, diff, sum
**Purpose**
Profile execution time for function

**GUI Alternatives**
As an alternative to the `profile` function, select **Desktop > Profiler** to open the Profiler.

**Syntax**

```matlab
profile on
profile -history
profile -nohistory
profile -history -historysize integer
profile -timer clock
profile -history -historysize integer -timer clock
profile off
profile resume
profile clear
profile viewer
S = profile('status')
stats = profile('info')
```

**Description**
The `profile` function helps you debug and optimize M-files by tracking their execution time. For each M-function, M-subfunction, or MEX-function in the file, `profile` records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Some people use `profile` simply to see the child functions; see also `depfun` for that purpose. To open the Profiler graphical user interface, use the `profile viewer` syntax. By default, Profiler time is CPU time. The total time reported by the Profiler is not the same as the time reported using the `tic` and `toc` functions or the time you would observe using a stopwatch.

**Note**
If your system uses Intel multi-core chips, you may want to restrict the active number of CPUs to 1 for the most accurate and efficient profiling. See “Intel Multi-Core Processors — Setting for Most Accurate Profiling” for details on how to do this.
profile on starts the Profiler, clearing previously recorded profile statistics. Note the following:

- You can specify all, none, or a subset, of the history, historysize and timer options with the profile on syntax.
- You can specify options in any order, including before or after on.
- If the Profiler is currently on and you specify profile with one of the options, MATLAB software returns an error message and the option has no effect. For example, if you specify profile timer real, MATLAB returns the following error: The profiler has already been started. TIMER cannot be changed.
- To change options, first specify profile off, and then specify profile on or profile resume with new options.

profile -history records the exact sequence of function calls. The profile function records, by default, up to 1,000,000 function entry and exit events. For more than 1,000,000 events, profile continues to record other profile statistics, but not the sequence of calls. To change the number of function entry and exit events that the profile function records, use the historysize option. By default, the history option is not enabled.

profile -nohistory disables further recording of the history (exact sequence of function calls). Use the -nohistory option after having previously set the -history option. All other profiling statistics continue to be collected.

profile -history -historysize integer specifies the number of function entry and exit events to record. By default, historysize is set to 1,000,000.

profile -timer clock specifies the type of time to use. Valid values for clock are:

- 'cpu' — The Profiler uses computer time (the default).
- 'real' — The Profiler uses wall-clock time.
For example, cpu time for the `pause` function is typically small, but real
time accounts for the actual time paused, and therefore would be larger.

```
profile -history -historysize integer -timer clock
```
specifies all
of the options. Any order is acceptable, as is a subset.

```
profile off
```
stops the Profiler.

```
profile resume
```
restarts the Profiler without clearing previously
recorded statistics.

```
profile clear
```
clears the statistics recorded by `profile`.

```
profile viewer
```
stops the Profiler and displays the results in the
Profiler window. For more information, see Profiling for Improving
Performance in the Desktop Tools and Development Environment
documentation.

```
S = profile('status')
```
returns a structure containing information
about the current status of the Profiler. The table lists the fields in the
order that they appear in the structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProfilerStatus</td>
<td>'on' or 'off'</td>
<td>'off'</td>
</tr>
<tr>
<td>DetailLevel</td>
<td>'mmex'</td>
<td>'mmex'</td>
</tr>
<tr>
<td>Timer</td>
<td>'cpu' or 'real'</td>
<td>'cpu'</td>
</tr>
<tr>
<td>HistoryTracking</td>
<td>'on' or 'off'</td>
<td>'off'</td>
</tr>
<tr>
<td>HistorySize</td>
<td>integer</td>
<td>1000000</td>
</tr>
</tbody>
</table>

```
stats = profile('info')
```
displays a structure containing the results.
Use this function to access the data generated by `profile`. The table
lists the fields in the order that they appear in the structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionTable</td>
<td>Structure array containing statistics about each function called</td>
</tr>
</tbody>
</table>
The FunctionTable field is an array of structures, where each structure contains information about one of the functions or subfunctions called during execution. The following table lists these fields in the order that they appear in the structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompleteName</td>
<td>Full path to FunctionName, including subfunctions</td>
</tr>
<tr>
<td>FunctionName</td>
<td>Function name; includes subfunctions</td>
</tr>
<tr>
<td>FileName</td>
<td>Full path to FunctionName, with file extension, excluding subfunctions</td>
</tr>
<tr>
<td>Type</td>
<td>M-functions, MEX-functions, and many other types of functions including M-subfunctions, nested functions, and anonymous functions</td>
</tr>
<tr>
<td>NumCalls</td>
<td>Number of times the function was called</td>
</tr>
<tr>
<td>TotalTime</td>
<td>Total time spent in the function and its child functions</td>
</tr>
<tr>
<td>TotalRecursiveTime</td>
<td>No longer used.</td>
</tr>
<tr>
<td>Children</td>
<td>FunctionTable indices to child functions</td>
</tr>
<tr>
<td>Parents</td>
<td>FunctionTable indices to parent functions</td>
</tr>
</tbody>
</table>
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutedLines</td>
<td>Array containing line-by-line details for the function being profiled. Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix. Column 2: Number of times the line was executed. Column 3: Total time spent on that line. Note: The sum of Column 3 entries does not necessarily add up to the function’s <code>TotalTime</code>.</td>
</tr>
<tr>
<td>IsRecursive</td>
<td>BOOLEAN value: Logical 1 (true) if recursive, otherwise logical 0 (false)</td>
</tr>
<tr>
<td>PartialData</td>
<td>BOOLEAN value: Logical 1 (true) if function was modified during profiling, for example by being edited or cleared. In that event, data was collected only up until the point when the function was modified.</td>
</tr>
</tbody>
</table>

### Examples

#### Profile and Display Results

This example profiles the MATLAB `magic` command and then displays the results in the Profiler window. The example then retrieves the profile data on which the HTML display is based and uses the `profsave` command to save the profile data in HTML form.

```matlab
profile on
plot(magic(35))
profile viewer
p = profile('info');
profsave(p,'profile_results')
```
Profile and Save Results

Another way to save profile data is to store it in a MAT-file. This example stores the profile data in a MAT-file, clears the profile data from memory, and then loads the profile data from the MAT-file. This example also shows a way to bring the reloaded profile data into the Profiler graphical interface as live profile data, not as a static HTML page.

```matlab
p = profile('info');
save myprofiledata p
clear p
load myprofiledata
profview(0,p)
```

Profile and Show Results Including History

This example illustrates an effective way to view the results of profiling when the history option is enabled. The history data describes the sequence of functions entered and exited during execution. The `profile` command returns history data in the `FunctionHistory` field of the structure it returns. The history data is a 2-by-

```matlab
profile on -history
plot(magic(4));
p = profile('info');

for n = 1:size(p.FunctionHistory,2)
    if p.FunctionHistory(1,n)==0
        str = 'entering function: ';
    else
        str = 'exiting function: ';
    end
```
disp([str p.FunctionTable(p.FunctionHistory(2,n)).FunctionName])
end

See Also  

depdir, depfun, mlint, profsave

Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation

“Using the Parallel Profiler” in the Parallel Computing Toolbox documentation
**Purpose**

Save profile report in HTML format

**Syntax**

```
profsave
profsave(profinfo)
profsave(profinfo,dirname)
```

**Description**

`profsave` executes the `profile('info')` function and saves the results in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of the structure returned by `profile`. By default, `profsave` stores the HTML files in a subdirectory of the current directory named `profile_results`.

`profsave(profinfo)` saves the profiling results, `profinfo`, in HTML format. `profinfo` is a structure of profiling information returned by the `profile('info')` function.

`profsave(profinfo,dirname)` saves the profiling results, `profinfo`, in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of `profinfo` and stores them in the directory specified by `dirname`.

**Examples**

Run profile and save the results.

```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

**See Also**

`profile`

Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**
Open Property Editor

**Syntax**
propedit
propedit(handle_list)

**Description**
propedit starts the Property Editor, a graphical user interface to the properties of graphics objects. If no current figure exists, propedit will create one.

propedit(handle_list) edits the properties for the object (or objects) in handle_list.

Starting the Property Editor enables plot editing mode for the figure.

**See Also**
inspect, plotedit, propertyeditor
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Open built-in property page for control</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>h.propedit</td>
</tr>
<tr>
<td></td>
<td>propedit(h)</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>h.propedit requests the control to display its built-in property page. Note that some controls do not have a built-in property page. For those controls, this command fails. propedit(h) is an alternate syntax for the same operation.</td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td>COM functions are available on Microsoft Windows systems only.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>Create a Microsoft Calendar control and display its property page:</td>
</tr>
<tr>
<td></td>
<td>cal = actxcontrol('mscal.calendar', [0 0 500 500]);</td>
</tr>
<tr>
<td></td>
<td>cal.propedit</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>inspect, get (COM)</td>
</tr>
</tbody>
</table>
**Purpose**

Display class property names

**Syntax**

```matlab
properties('classname')
properties(obj)
p = properties(...)```

**Description**

`properties('classname')` displays the names of the public properties for the MATLAB class `classname`, including public properties inherited from superclasses.

`properties(obj)` displays the names of the public properties for the class of the object `obj`, where `obj` is an instance of a MATLAB class. `obj` can be either a scalar object or an array of objects. When `obj` is scalar, `properties` also returns dynamic properties.

See “Dynamic Properties — Adding Properties to an Instance” for information on using dynamic properties.

`p = properties(...)` returns the property names in a cell array of strings. Note that you can use the Workspace browser to browse current property values. See “MATLAB Workspace” for more information on using the Workspace browser.

A property is public when its `GetAccess` attributes are set to `public` and its `Hidden` attribute is set to `false` (default values for these attributes). See “Property Attributes” for a complete list of attributes.

You can also use the `fieldnames` function to list property names of MATLAB classes.

**Note**

`properties` is also a keyword used in MATLAB class definition. See `classdef` for more information on class definition keywords.

See “Properties — Storing Class Data” for more information on class properties.
**Examples**

Retrieve the names of the public properties of class `memmapfile` and store the result in a cell array of strings:

```matlab
p = properties('memmapfile');
p
ans =

'writable'
'offset'
'format'
'repeat'
'filename'
```

Construct an instance of the `MException` class and get its properties names:

```matlab
me = MException('Msg:ID','MsgText');
properties(me)
Properties for class MException:

    identifier
    message
    cause
    stack
```

**See Also**

`events`, `fieldnames`, `methods`
**Purpose**
Show or hide property editor

**GUI Alternatives**
Click the larger Plotting Tools icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Open or close the Property Editor tool from the figure’s View menu. For details, see “The Property Editor” in the MATLAB Graphics documentation.

**Syntax**
```matlab
propertyeditor('on')
propertyeditor('off')
propertyeditor('toggle')
propertyeditor(figure_handle,...)
```

**Description**
- `propertyeditor('on')` displays the Property Editor on the current figure.
- `propertyeditor('off')` hides the Property Editor on the current figure.
- `propertyeditor('toggle')` or `propertyeditor` toggles the visibility of the property editor on the current figure.
- `propertyeditor(figure_handle,...)` displays or hides the Property Editor on the figure specified by `figure_handle`.

**See Also**
plottools, plotbrowser, figurepalette, inspect
**Purpose**
Psi (polygamma) function

**Syntax**
- \( Y = \psi(X) \)
- \( Y = \psi(k,X) \)
- \( Y = \psi(k0:k1,X) \)

**Description**
- \( Y = \psi(X) \) evaluates the \( \Psi \) function for each element of array \( X \). \( X \) must be real and nonnegative. The \( \Psi \) function, also known as the digamma function, is the logarithmic derivative of the gamma function.

\[
\psi(x) = \frac{d}{dx} \log(\Gamma(x)) = \frac{d\Gamma(x)}{dx} \cdot \frac{1}{\Gamma(x)}
\]

- \( Y = \psi(k,X) \) evaluates the \( k \)th derivative of \( \Psi \) at the elements of \( X \).
- \( \psi(0,X) \) is the digamma function, \( \psi(1,X) \) is the trigamma function, \( \psi(2,X) \) is the tetragamma function, etc.
- \( Y = \psi(k0:k1,X) \) evaluates derivatives of order \( k0 \) through \( k1 \) at \( X \).
- \( Y(k,j) \) is the \( (k-1+k0) \)th derivative of \( \Psi \), evaluated at \( X(j) \).

**Examples**

**Example 1**
Use the \( \psi \) function to calculate Euler's constant, \( \gamma \).

```matlab
format long
-psi(1)
ans =
    0.57721566490153

-psi(0,1)
ans =
    0.57721566490153
```
Example 2

The trigamma function of 2, \( \psi(1,2) \), is the same as \( \left( \frac{\pi^2}{6} \right) - 1 \).

```matlab
format long
psi(1,2)
ans =
0.64493406684823

pi^2/6 - 1
ans =
0.64493406684823
```

Example 3

This code produces the first page of Table 6.1 in Abramowitz and Stegun [1].

```matlab
x = (1:.005:1.250)';
[x gamma(x) gammaln(x) psi(0:1,x)' x-1]
```

Example 4

This code produces a portion of Table 6.2 in [1].

```matlab
psi(2:3,1:.01:2)'
```

See Also

gamma, gammainc, gammaln

References

Purpose
Publish M-file containing cells, saving output to file of specified type

GUI
As an alternative to the publish function, use the File > Publish filename menu or File > Publish Configuration for filename items in the Editor.

Syntax
publish('script')
publish('script','format')
publish('script', options)
publish('function', options)

Description
publish('script') runs the M-file script named script in the base workspace one cell at a time, and saves the code, comments, and results to an HTML output file. The output file is named script.html and is stored, along with other supporting output files, in an html subdirectory in script's directory.

publish('script','format') runs the M-file script named script, one cell at a time in the base workspace, and publishes the code, comments, and results to an output file using the specified format. Allowable values for format are html (the default), xml, latex for LaTeX, doc for Microsoft Word documents, and ppt for Microsoft PowerPoint documents. The output file is named script.format and is stored, along with other supporting output files, in an html subdirectory in script's directory. The doc format requires the Word application, and the ppt format requires PowerPoint application. When publishing to HTML, the M-file code is included at the end of published HTML file as comments, even when the showCode option is set to false. Because it is included as comments, it does not display in a Web browser. Use the grabcode function to extract the code from the HTML file.

publish('script', options) publishes using the structure options, which can contain any of the fields and corresponding value for each field as shown in Options for publish on page 2-2903. Create and save structures for the options you use regularly. For details, see “Specify Values for the Publish Settings Property Table” in the online documentation for MATLAB software.
publish('function', options) publishes an M-file function using the structure options. The codeToEvaluate field must specify the function input and the file to publish if you set the evalCode field to true. If you set the evalCode field to false, it essentially saves the M-file to another format, such as HTML, which allows display with formatting in a Web browser.

**Options for publish**

<table>
<thead>
<tr>
<th>Field</th>
<th>Allowable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>'doc', 'html' (default), 'latex', 'ppt', 'xml'</td>
</tr>
<tr>
<td>stylesheet</td>
<td>'' (default), XSL file name (used only when format is html, latex, or xml)</td>
</tr>
<tr>
<td>outputDir</td>
<td>'' (default, a subfolder named html), full path</td>
</tr>
<tr>
<td>imageFormat</td>
<td>'png' (default unless format is latex), 'epsc2' (default when format is latex), any format supported by print when figureSnapMethod is print, any format supported by imwrite functions when figureSnapMethod is getframe, entireFigureWindow, or entireGUIWindow.</td>
</tr>
<tr>
<td>figureSnapMethod</td>
<td>'entireGUIWindow' (default), 'entireFigureWindow', 'print', 'getframe'</td>
</tr>
<tr>
<td>showCode</td>
<td>true (default), false</td>
</tr>
<tr>
<td>evalCode</td>
<td>true (default), false</td>
</tr>
</tbody>
</table>

See the FigureSnap Method Options table for details on the effects of these settings.
### Options for publish (Continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Allowable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>catchError</td>
<td>true (default, continues publishing and includes the error in the published file), false (displays the error and publishing ends)</td>
</tr>
<tr>
<td>codeToEvaluate</td>
<td>M-file you are publishing (default), any valid code</td>
</tr>
<tr>
<td>createThumbnail</td>
<td>true (default), false</td>
</tr>
<tr>
<td>maxOutputLines</td>
<td>Inf (default), nonnegative integer specifying the maximum number of output lines to publish per M-file cell before truncating the output</td>
</tr>
</tbody>
</table>

### FigureSnapMethod Options

<table>
<thead>
<tr>
<th>To Get Figure Captures with These Appearance Details</th>
<th>Use this FigureSnapMethod Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window Decorations</td>
<td></td>
</tr>
<tr>
<td>Included for dialog boxes; Excluded for figures</td>
<td></td>
</tr>
<tr>
<td>Plot Backgrounds</td>
<td></td>
</tr>
<tr>
<td>Set to white for figures; match screen for dialog boxes.</td>
<td>entireGUIWindow</td>
</tr>
<tr>
<td>Excluded for dialog boxes and figures</td>
<td>All set to white</td>
</tr>
<tr>
<td>Included for dialog boxes and figures</td>
<td>All match screen</td>
</tr>
<tr>
<td>Excluded for dialog boxes and figures</td>
<td>All match screen</td>
</tr>
<tr>
<td>Included for dialog boxes and figures</td>
<td></td>
</tr>
</tbody>
</table>
Remarks

Be aware that publish opens a Figure window and sets the background color to white when publishing begins. It closes this Figure window when publishing ends. If your M-code explicitly creates one or more additional Figure windows, publish uses the default background color (gray) unless you specify otherwise. For example, if you publish the following code (using default option settings), the first and second plots publish with a white background and the third plot publishes with a gray background.

```matlab
%% Plot 1
plot(1:10)

%% Plot 2
plot(5:5:25)

%% Plot 3
f = figure;
plot(1:10)
```

Examples

To use the M-Files referenced in the following examples, open the files using the following commands and save each file to a directory to which you have write access. In each example, it is assumed that the directory to which you save the files is `I:/my_matlab_files/my_mfiles`, so you may need to adjust accordingly.

```matlab
edit(fullfile(matlabroot,'help','techdoc','matlab_env','examples','
edit(fullfile(matlabroot,'help','techdoc','matlab_env','examples','
```

Publish to HTML Format

To publish the M-file script `I:/myfiles/sine_wave.m` to HTML, run the following command:

```matlab
publish('I:/my_matlab_files/my_mfiles/sine_wave.m', 'html')
web('I:/my_matlab_files/my_mfiles/html/sine_wave.html')
```

MATLAB runs the file and saves the code, comments, and results to `I:/myfiles/html/sine_wave.html`. Open that file in the Web browser to view the published output.
Publish to Microsoft Word Format

This example defines the structure options_doc_nocode, publishes sine_wave.m using the defined options, and displays the resulting document. The resulting report is a Microsoft Word document, I:/my_doc_files/sine_wave.doc and includes results, but not the MATLAB code.

```matlab
options_doc_nocode.format='doc'
options_doc_nocode.outputDir='I:/my_doc_files'
options_doc_nocode.showCode=false
publish('I:/my_matlab_files/my_mfiles/sine_wave.m',options_doc_nocode)
winopen('I:/my_doc_files/sine_wave.doc')
```

Publish Function M-File and Evaluate Code

This example defines the structure function_options, which specifies the value of the input argument to the function, publishes the function I:/my_matlab_files/my_mfiles/collatz.m, and displays the resulting HTML document, I:/my_matlab_files/my_mfiles/html/collatz.html:

```matlab
function_options.format='html'
function_options.evalCode=true;
function_options.codeToEvaluate=[ ...
' n=3' char(10) ...
' collatz(3)' char(10) ...
]
function_options.showCode=true;
publish('I:/my_matlab_files/my_mfiles/collatz.m',function_options);
web('I:/my_matlab_files/my_mfiles/html/collatz.html')
```

Publish M-File Script and Capture Window Decorations

This example defines the structure function_options, publishes the function I:/my_matlab_files/my_mfiles/sine_wave.m, and displays the resulting HTML document, I:/my_matlab_files/my_mfiles/html/sine_wave.html:
function_options.format='html';
function_options.figureSnapMethod='entireGUIWindow';
publish('I:/my_matlab_files/my_mfiles/sine_wave.m',function_options);
web('I:/my_matlab_files/my_mfiles/html/sine_wave.html')

See Also

grabcode, notebook, web, winopen

MATLAB Desktop Tools and Development Environment documentation, specifically:

• “Defining Cells”
• “Publishing M-Files”
**PutCharArray**

**Purpose**
Store character array in Automation server

**Syntax**

**MATLAB Client**

```matlab
h.PutCharArray('varname', 'workspace', 'string')
PutCharArray(h, 'varname', 'workspace', 'string')
invoke(h, 'PutCharArray', 'varname', 'workspace', 'string')
```

**Method Signature**


**Microsoft Visual Basic Client**

```vbnet
PutCharArray(varname As String, workspace As String, string As String)
```

**Description**

PutCharArray stores the character array in string in the specified workspace of the server attached to handle h, assigning to it the variable varname. The workspace argument can be either base or global.

**Remarks**

The character array specified in the string argument can have any dimensions. However, PutCharArray changes the dimensions to a 1–by-n column-wise representation, where n is the number of characters in the array. Executing the following commands in MATLAB illustrates this behavior:

```matlab
h = actxserver('matlab.application');
chArr = [ 'abc'; 'def'; 'ghk']
chArr =
  abc
def
  ghk
h.PutCharArray('Foo', 'base', chArr)
tstArr = h.GetCharArray('Foo', 'base')
tstArr =
adgbehcfk
```
Server function names, like `PutCharArray`, are case sensitive when using the dot notation syntax shown in the Syntax section.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

**Examples**

Store string `str` in the base workspace of the server using `PutCharArray`.

**MATLAB Client**

```matlab
h = actxserver('matlab.application');
h.PutCharArray('str', 'base', ...
              'He jests at scars that never felt a wound.');
S = h.GetCharArray('str', 'base')
S =
    He jests at scars that never felt a wound.
```

**Visual Basic .NET Client**

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```vbasic
Dim Matlab As Object
Try
    Matlab = GetObject(, "matlab.application")
Catch e As Exception
    Matlab = CreateObject("matlab.application")
End Try
MsgBox("MATLAB window created; now open it...")

Open the MATLAB window, then click `Ok`.

Matlab.PutCharArray("str", "base", _
    "He jests at scars that never felt a wound.")
MsgBox("In MATLAB, type" & vbCrLf _
    & "str")
```
In the MATLAB window type `str`; MATLAB displays:

```matlab
str = 
He jests at scars that never felt a wound.
```

Click **Ok**.

```matlab
MsgBox("closing MATLAB window...")
```

Click **Ok** to close and terminate MATLAB.

```matlab
Matlab.Quit()
```

**See Also**

GetCharArray, PutWorkspaceData, GetWorkspaceData, Execute
**Purpose**  
Store matrix in Automation server

**Syntax**

**MATLAB Client**

```matlab
h.PutFullMatrix('varname', 'workspace', xreal, ximag)
PutFullMatrix(h, 'varname', 'workspace', xreal, ximag)
invoke(h, 'PutFullMatrix', 'varname', 'workspace', xreal, ximag)
```

**Method Signature**

```c
PutFullMatrix([in] BSTR varname, [in] BSTR workspace,
[in] SAFEARRAY(double) xreal, [in] SAFEARRAY(double) ximag)
```

**Microsoft Visual Basic Client**

```vbs
PutFullMatrix([in] varname As String, [in] workspace As String,
[in] xreal As Double, [in] ximag As Double)
```

**Description**

PutFullMatrix stores a matrix in the specified workspace of the server attached to handle h, assigning to it the variable varname. Enter the real and imaginary parts of the matrix in the xreal and ximag input arguments. The workspace argument can be either base or global.

**Remarks**

The matrix specified in the xreal and ximag arguments cannot be scalar, an empty array, or have more than two dimensions.

Server function names, like PutFullMatrix, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

For VBScript clients, use the GetWorkspaceData and PutWorkspaceData functions to pass numeric data to and from the MATLAB workspace. These functions use the variant data type instead of SAFEARRAY which is not supported by VBScript.
Examples Writing to the Base Workspace Example

Assign a 5-by-5 real matrix to the variable \( M \) in the base workspace of the server, and then read it back with \( \text{GetFullMatrix} \). The real and imaginary parts are passed in through separate arrays of doubles.

**MATLAB Client**

\[
\begin{align*}
\text{h} &= \text{actxserver('matlab.application')}; \\
\text{h}.\text{PutFullMatrix}('M', 'base', \text{rand}(5), \text{zeros}(5)) \\
% \text{One output returns real, use two for real and imag} \\
\text{xreal} &= \text{h}.\text{GetFullMatrix}('M', 'base', \text{zeros}(5), \text{zeros}(5)) \\
\text{xreal} &= \\
0.9501 & 0.7621 & 0.6154 & 0.4057 & 0.0579 \\
0.2311 & 0.4565 & 0.7919 & 0.9355 & 0.3529 \\
0.6068 & 0.0185 & 0.9218 & 0.9169 & 0.8132 \\
0.4860 & 0.8214 & 0.7382 & 0.4103 & 0.0099 \\
0.8913 & 0.4447 & 0.1763 & 0.8936 & 0.1389
\end{align*}
\]

**Visual Basic .NET Client**

\[
\begin{align*}
\text{Dim MatLab} & \text{ As Object} \\
\text{Dim XReal}(4, 4) & \text{ As Double} \\
\text{Dim XImag}(4, 4) & \text{ As Double} \\
\text{Dim ZReal}(4, 4) & \text{ As Double} \\
\text{Dim ZImag}(4, 4) & \text{ As Double} \\
\text{Dim i, j} & \text{ As Integer} \\
\text{For i = 0 To 4} \\
\text{\quad For j = 0 To 4} \\
\text{\quad XReal(i, j) = Rnd() * 6} \\
\text{\quad XImag(i, j) = 0} \\
\text{\quad Next j} \\
\text{Next i} \\
\text{Matlab} & = \text{CreateObject("matlab.application")]} \\
\text{MatLab.PutFullMatrix("M", "base", XReal, XImag)} \\
\text{MatLab.GetFullMatrix("M", "base", ZReal, ZImag)}
\end{align*}
\]
**Writing to the Global Workspace Example**

Write a matrix to the global workspace of the server and then examine the server's global workspace from the client.

**MATLAB Client**

```matlab
h = actxserver('matlab.application');
h.PutFullMatrix('X', 'global', [1 3 5; 2 4 6], ...
                [1 1 1; 1 1 1])
h.invoke('Execute', 'whos global')
ans =
    Name      Size       Bytes  Class          Attributes
    X       2x3         96 bytes double array (global complex)
Grand total is 6 elements using 96 bytes
```

**Visual Basic .NET Client**

```vbnet
Dim MatLab As Object
Dim XReal(1, 2) As Double
Dim XImag(1, 2) As Double
Dim result As String
Dim i, j As Integer

For i = 0 To 1
    For j = 0 To 2
        XReal(i, j) = (j * 2 + 1) + i
        XImag(i, j) = 1
    Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("X", "global", XReal, XImag)
result = Matlab.Execute("whos global")
MsgBox(result)
```
PutFullMatrix

See Also

GetFullMatrix, PutWorkspaceData, GetWorkspaceDataExecute
**Purpose**
Store data in Automation server workspace

**Syntax**

**MATLAB Client**

```matlab
h.PutWorkspaceData('varname', 'workspace', data)
PutWorkspaceData(h, 'varname', 'workspace', data)
invoke(h, 'PutWorkspaceData', 'varname', 'workspace', data)
```

**Method Signature**


**Microsoft Visual Basic Client**

```vb
PutWorkspaceData(varname As String, workspace As String, data As Object)
```

**Description**

PutWorkspaceData stores data in the specified workspace of the server attached to handle `h`, assigning to it the variable `varname`. The workspace argument can be either base or global.

**Note**

PutWorkspaceData works on all MATLAB types except sparse arrays, structure arrays, and function handles. Use the Execute method for these data types.

**Passing Character Arrays**

MATLAB enables you to define 2-D character arrays such as the following:

```matlab
chArr = ['abc';'def';'ghk']
chArr =
abc
def
ghk
size(chArr)
ans =
    3   3
```
However, `PutWorkspaceData` does not preserve the dimensions of character arrays when passing them to a COM server. 2-D arrays are converted to 1-by-n arrays of characters, where n equals the number of characters in the original array plus one newline character for each row in the original array. This means that `chArr` above is converted to a 1-by-12 array, but the newline characters make it display with three rows in the MATLAB command window. For example:

```matlab
h = actxserver('matlab.application');
h.PutWorkspaceData('Foo','base',chArr);
tstArr = h.GetWorkspaceData('Foo','base')
tstArr =
    abc
def
   ghk

size(tstArr)
ans =
    1    12
```

**Remarks**

You can use `PutWorkspaceData` in place of `PutFullMatrix` and `PutCharArray` to pass numeric and character array data respectively to the server.

Server function names, like `PutWorkspaceData`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

The `GetWorkspaceData` and `PutWorkspaceData` functions pass numeric data as a variant data type. These functions are especially useful for VBScript clients as VBScript does not support the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`.

**Examples**

Create an array in the client and assign it to variable `A` in the base workspace of the server:
**MATLAB Client**

```matlab
h = actxserver('matlab.application');
for i = 0:6
    data(i+1) = i * 15;
end
h.PutWorkspaceData('A', 'base', data)
```

**Visual Basic .NET Client**

This example uses the Visual Basic `MsgBox` command to control flow between MATLAB and the Visual Basic Client.

```vba
Dim Matlab As Object
Dim data(6) As Double
Dim i As Integer
MatLab = CreateObject("matlab.application")
For i = 0 To 6
    data(i) = i * 15
Next i
MatLab.PutWorkspaceData("A", "base", data)
MsgBox("In MATLAB, type" & vbCrLf & "A")
```

Open the MATLAB window and type `A`. MATLAB displays:

```
A =
    0   15   30   45   60   75   90
```

Click **Ok** to close and terminate MATLAB.

**See Also**

- GetWorkspaceData
- PutFullMatrix
- GetFullMatrix
- PutCharArray
- GetCharArrayExecute

See “Introduction” for more examples.
### Purpose
Identify current directory

### Graphical Interface
As an alternative to the `pwd` function, you can use the current directory field on the desktop toolbar or the Current Directory browser.

### Syntax
```
pwd
s = pwd
```

### Description
`pwd` displays the current working directory.

`s = pwd` returns the current directory to the variable `s`.

On Microsoft Windows platforms, to go directly to the current working directory, use
```
winopen(pwd)
```

### See Also
`cd`, `dir`, `fileparts`, `mfilename`, `path`, `what`, `winopen`

“Managing Files and Working with the Current Directory”
**Purpose**  
Quasi-minimal residual method

**Syntax**
```matlab
x = qmr(A,b)
qmr(A,b,tol)
qmr(A,b,tol,maxit)
qmr(A,b,tol,maxit,M)
qmr(A,b,tol,maxit,M1,M2)
qmr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = qmr(A,b,...)
[x,flag,relres] = qmr(A,b,...)
[x,flag,relres,iter] = qmr(A,b,...)
[x,flag,relres,iter,resvec] = qmr(A,b,...)
```

**Description**

\( x = \text{qmr}(A,b) \) attempts to solve the system of linear equations \( A*x=b \) for \( x \). The \( n \)-by-\( n \) coefficient matrix \( A \) must be square and should be large and sparse. The column vector \( b \) must have length \( n \). \( A \) can be a function handle \( \text{afun} \) such that \( \text{afun}(x,'\text{notransp}') \) returns \( A*x \) and \( \text{afun}(x,'\text{transp}') \) returns \( A'*x \). See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function \( \text{afun} \), as well as the preconditioner function \( \text{mfun} \) described below, if necessary.

If \( \text{qmr} \) converges, a message to that effect is displayed. If \( \text{qmr} \) fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual \( \text{norm}(b-A*x)/\text{norm}(b) \) and the iteration number at which the method stopped or failed.

\( \text{qmr}(A,b,tol) \) specifies the tolerance of the method. If \( tol \) is [], then \( \text{qmr} \) uses the default, \( 1e^{-6} \).

\( \text{qmr}(A,b,tol,maxit) \) specifies the maximum number of iterations. If \( maxit \) is [], then \( \text{qmr} \) uses the default, \( \text{min}(n,20) \).

\( \text{qmr}(A,b,tol,maxit,M) \) and \( \text{qmr}(A,b,tol,maxit,M1,M2) \) use preconditioners \( M \) or \( M = M1*M2 \) and effectively solve the system.
\[
\text{inv}(M)*A*x = \text{inv}(M)*b \quad \text{for} \quad x.
\]
If \( M \) is \( [ ] \) then \text{qmr} applies no preconditioner. \( M \) can be a function handle \( \text{mfun} \) such that \( \text{mfun}(x,'\text{notransp}') \) returns \( M\backslash x \) and \( \text{mfun}(x,'\text{transp}') \) returns \( M'\backslash x \).

\text{qmr}(A,b,\text{tol},\text{maxit},M_1,M_2,x0) \) specifies the initial guess. If \( x0 \) is \( [ ] \), then \text{qmr} uses the default, an all zero vector.

\([x,\text{flag}] = \text{qmr}(A,b,...)\) also returns a convergence flag.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\text{qmr} converged to the desired tolerance \text{tol} within \text{maxit} iterations.</td>
</tr>
<tr>
<td>1</td>
<td>\text{qmr} iterated \text{maxit} times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>Preconditioner ( M ) was ill-conditioned.</td>
</tr>
<tr>
<td>3</td>
<td>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during \text{qmr} became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

Whenever \text{flag} is not \( 0 \), the solution \( x \) returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the \text{flag} output is specified.

\([x,\text{flag},\text{relres}] = \text{qmr}(A,b,...)\) also returns the relative residual \( \text{norm}(b-A*x)/\text{norm}(b) \). If \text{flag} is \( 0 \), \( \text{relres} \leq \text{tol} \).

\([x,\text{flag},\text{relres},\text{iter}] = \text{qmr}(A,b,...)\) also returns the iteration number at which \( x \) was computed, where \( 0 \leq \text{iter} \leq \text{maxit} \).

\([x,\text{flag},\text{relres},\text{iter},\text{resvec}] = \text{qmr}(A,b,...)\) also returns a vector of the residual norms at each iteration, including \( \text{norm}(b-A*x0) \).

**Examples**

**Example 1**

\[
n = 100; \\
on = \text{ones}(n,1); \\
\]
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8; maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = qmr(A,b,tol,maxit,M1,M2);

displays the message

qmr converged at iteration 9 to a solution...
with relative residual
5.6e-009

Example 2

This example replaces the matrix A in Example 1 with a handle to a matrix-vector product function afun. The example is contained in an M-file run_qmr that

- Calls qmr with the function handle @afun as its first argument.
- Contains afun as a nested function, so that all variables in run_qmr are available to afun.

The following shows the code for run_qmr:

```matlab
function x1 = run_qmr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = qmr(@afun,b,tol,maxit,M1,M2);

function y = afun(x,transp_flag)
    if strcmp(transp_flag,'transp') % y = A'*x
```
```matlab
y = 4 * x;
    y(1:n-1) = y(1:n-1) - 2 * x(2:n);
    y(2:n) = y(2:n) - x(1:n-1);
elseif strcmp(transp_flag,'notransp') % y = A*x
    y = 4 * x;
    y(2:n) = y(2:n) - 2 * x(1:n-1);
    y(1:n-1) = y(1:n-1) - x(2:n);
end
end

When you enter

```matlab
x1=run_qmr;
```matlab
MATLAB software displays the message

```matlab
qmr converged at iteration 9 to a solution with relative residual 5.6e-009
```matlab

**Example 3**

```matlab
load west0479;
A = west0479;
b = sum(A,2);
[x,flag] = qmr(A,b)
```

`flag` is 1 because `qmr` does not converge to the default tolerance 1e-6 within the default 20 iterations.

```matlab
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = qmr(A,b,1e-6,20,L1,U1)
```

`flag1` is 2 because the upper triangular `U1` has a zero on its diagonal, and `qmr` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` using backslash.

```matlab
[L2,U2] = luinc(A,1e-6);
[x2,flag2,relres2,iter2,resvec2] = qmr(A,b,1e-15,10,L2,U2)
```
flag2 is 0 because qmr converges to the tolerance of $1.6571\times 10^{-16}$ (the value of relres2) at the eighth iteration (the value of iter2) when preconditioned by the incomplete LU factorization with a drop tolerance of $1e^{-6}$. resvec2(1) = norm(b) and resvec2(9) = norm(b - A*x2).

You can follow the progress of qmr by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```matlab
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```

See Also
bicg, bicgstab, cgs, gmres, lsqr, luinc, minres, pcg, symmlq, function_handle (@), mldivide (\)
References


Purpose

Orthogonal-triangular decomposition

Syntax

\[
\begin{align*}
[Q,R] &= \text{qr}(A) \quad \text{(full and sparse matrices)} \\
[Q,R] &= \text{qr}(A,0) \quad \text{(full and sparse matrices)} \\
[Q,R,E] &= \text{qr}(A) \quad \text{(full matrices)} \\
[Q,R,E] &= \text{qr}(A,0) \quad \text{(full matrices)} \\
X &= \text{qr}(A) \quad \text{(full matrices)} \\
R &= \text{qr}(A) \quad \text{(sparse matrices)} \\
[C,R] &= \text{qr}(A,B) \quad \text{(sparse matrices)} \\
R &= \text{qr}(A,0) \quad \text{(sparse matrices)} \\
[C,R] &= \text{qr}(A,B,0) \quad \text{(sparse matrices)}
\end{align*}
\]

Description

The `qr` function performs the orthogonal-triangular decomposition of a matrix. This factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of a real complex unitary matrix and an upper triangular matrix.

\[ [Q,R] = \text{qr}(A) \] produces an upper triangular matrix \( R \) of the same dimension as \( A \) and a unitary matrix \( Q \) so that \( A = Q*R \). For sparse matrices, \( Q \) is often nearly full. If \([m \ n] = \text{size}(A)\), then \( Q \) is \( m \)-by-\( m \) and \( R \) is \( m \)-by-\( n \).

\[ [Q,R] = \text{qr}(A,0) \] produces an “economy-size” decomposition. If \([m \ n] = \text{size}(A)\), and \( m > n \), then `qr` computes only the first \( n \) columns of \( Q \) and \( R \) is \( n \)-by-\( n \). If \( m \leq n \), it is the same as \( [Q,R] = \text{qr}(A) \).

\[ [Q,R,E] = \text{qr}(A) \] for full matrix \( A \), produces a permutation matrix \( E \), an upper triangular matrix \( R \) with decreasing diagonal elements, and a unitary matrix \( Q \) so that \( A*E = Q*R \). The column permutation \( E \) is chosen so that \( \text{abs(diag}(R)) \) is decreasing.

\[ [Q,R,E] = \text{qr}(A,0) \] for full matrix \( A \), produces an “economy-size” decomposition in which \( E \) is a permutation vector, so that \( A(:,E) = Q*R \). The column permutation \( E \) is chosen so that \( \text{abs(diag}(R)) \) is decreasing.

\( X = \text{qr}(A) \) for full matrix \( A \), returns the output of the LAPACK subroutine DGEQRF or ZGEQRF. \( \text{triu(qr}(A)) \) is \( R \).
R = qr(A) for sparse matrix A, produces only an upper triangular matrix, R. The matrix R provides a Cholesky factorization for the matrix associated with the normal equations,

\[ R' \cdot R = A' \cdot A \]

This approach avoids the loss of numerical information inherent in the computation of \( A' \cdot A \). It may be preferred to \([Q,R] = qr(A)\) since Q is always nearly full.

\([C,R] = qr(A,B)\) for sparse matrix A, applies the orthogonal transformations to B, producing \( C = Q' \cdot B \) without computing Q. B and A must have the same number of rows.

R = qr(A,0) and \([C,R] = qr(A,B,0)\) for sparse matrix A, produce “economy-size” results.

For sparse matrices, the Q-less QR factorization allows the solution of sparse least squares problems

\[
\text{minimize} \| A x - b \|
\]

with two steps

\[
[C,R] = qr(A,b)
\]

\[
x = R \backslash C
\]

If A is sparse but not square, MATLAB software uses the two steps above for the linear equation solving backslash operator, i.e., \( x = A \backslash b \).

**Examples**

**Example 1**

Start with

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{bmatrix}
\]
This is a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization:

\[ [Q, R] = \text{qr}(A) \]

\[ Q = \begin{bmatrix}
-0.0776 & -0.8331 & 0.5444 & 0.0605 \\
-0.3105 & -0.4512 & -0.7709 & 0.3251 \\
-0.5433 & -0.0694 & -0.0913 & -0.8317 \\
-0.7762 & 0.3124 & 0.3178 & 0.4461
\end{bmatrix} \]

\[ R = \begin{bmatrix}
-12.8841 & -14.5916 & -16.2992 & \\
0 & -1.0413 & -2.0826 & \\
0 & 0 & 0.0000 & \\
0 & 0 & 0 &
\end{bmatrix} \]

The triangular structure of \( R \) gives it zeros below the diagonal; the zero on the diagonal in \( R(3,3) \) implies that \( R \), and consequently \( A \), does not have full rank.

**Example 2**

This example uses matrix \( A \) from the first example. The QR factorization is used to solve linear systems with more equations than unknowns. For example, let

\[ b = [1; 3; 5; 7] \]

The linear system \( Ax = b \) represents four equations in only three unknowns. The best solution in a least squares sense is computed by

\[ x = A\backslash b \]

which produces

Warning: Rank deficient, rank = 2, tol = 1.4594E-014
The quantity $\text{tol}$ is a tolerance used to decide if a diagonal element of $R$ is negligible. If $[Q,R,E] = \text{qr}(A)$, then
\[
\text{tol} = \max(\text{size}(A)) \cdot \text{eps} \cdot \text{abs}(R(1,1))
\]
The solution $x$ was computed using the factorization and the two steps
\[
y = Q' \cdot b;
\]
\[
x = R \cdot y
\]
The computed solution can be checked by forming $A \cdot x$. This equals $b$ to within roundoff error, which indicates that even though the simultaneous equations $A \cdot x = b$ are overdetermined and rank deficient, they happen to be consistent. There are infinitely many solution vectors $x$; the QR factorization has found just one of them.

**Algorithm**

**Inputs of Type Double**

For inputs of type `double`, `qr` uses the LAPACK routines listed in the following table to compute the QR decomposition.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Real</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = \text{qr}(A)$</td>
<td>DGEQRF</td>
<td>ZGEQRF</td>
</tr>
<tr>
<td>$X = \text{qr}(A,0)$</td>
<td>DGEQRF</td>
<td>ZGEQRF</td>
</tr>
<tr>
<td>$[Q,R] = \text{qr}(A)$</td>
<td>DGEQRF, DORGQR</td>
<td>ZGEQRF, ZUNGQR</td>
</tr>
<tr>
<td>$[Q,R] = \text{qr}(A,0)$</td>
<td>DGEQRF, DORGQR</td>
<td>ZGEQRF, ZUNGQR</td>
</tr>
<tr>
<td>$[Q,R,e] = \text{qr}(A)$</td>
<td>DGEQP3, DORGQR</td>
<td>ZGEQP3, ZUNGQR</td>
</tr>
<tr>
<td>$[Q,R,e] = \text{qr}(A,0)$</td>
<td>DGEQP3, DORGQR</td>
<td>ZGEQP3, ZUNGQR</td>
</tr>
</tbody>
</table>
Inputs of Type Single

For inputs of type single, qr uses the LAPACK routines listed in the following table to compute the QR decomposition.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Real</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R = qr(A)$</td>
<td>SGEQRF</td>
<td>CGEQRF</td>
</tr>
<tr>
<td>$R = qr(A,0)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[Q,R] = qr(A)$</td>
<td>SGEQRF, SORGQR</td>
<td>CGEQRF, CUNGQR</td>
</tr>
<tr>
<td>$[Q,R] = qr(A,0)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[Q,R,e] = qr(A)$</td>
<td>SGEQP3, SORGQR</td>
<td>CGEQP3, CUNGQR</td>
</tr>
<tr>
<td>$[Q,R,e] = qr(A,0)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Also

lu, null, orth, qrdelete, qrintert, qrupdate

The arithmetic operators \ and /

References

**Purpose**
Remove column or row from QR factorization

**Syntax**

\[
\begin{align*}
[Q1,R1] &= \text{qrdelete}(Q,R,j) \\
[Q1,R1] &= \text{qrdelete}(Q,R,j,\text{'col'}) \\
[Q1,R1] &= \text{qrdelete}(Q,R,j,\text{'row'})
\end{align*}
\]

**Description**

\[ [Q1,R1] = \text{qrdelete}(Q,R,j) \]
returns the QR factorization of the matrix \( A1 \), where \( A1 \) is \( A \) with the column \( A(:,j) \) removed and \([Q,R] = \text{qr}(A)\) is the QR factorization of \( A \).

\[ [Q1,R1] = \text{qrdelete}(Q,R,j,\text{'col'}) \]
is the same as \( \text{qrdelete}(Q,R,j) \).

\[ [Q1,R1] = \text{qrdelete}(Q,R,j,\text{'row'}) \]
returns the QR factorization of the matrix \( A1 \), where \( A1 \) is \( A \) with the row \( A(j,:) \) removed and \([Q,R] = \text{qr}(A)\) is the QR factorization of \( A \).

**Examples**

\[
\begin{align*}
A &= \text{magic}(5); \\
[Q,R] &= \text{qr}(A); \\
j &= 3; \\
[Q1,R1] &= \text{qrdelete}(Q,R,j,\text{'row'});
\end{align*}
\]

\[
\begin{align*}
Q1 &= \\
&= \begin{bmatrix}
0.5274 & -0.5197 & -0.6697 & -0.0578 \\
0.7135 & 0.6911 & 0.0158 & 0.1142 \\
0.3102 & -0.1982 & 0.4675 & -0.8037 \\
0.3413 & -0.4616 & 0.5768 & 0.5811
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
R1 &= \\
&= \begin{bmatrix}
& 0 & -19.7045 & -10.9891 & 0.4318 & -1.4873 \\
& 0 & 0 & 22.7444 & 5.8357 & -3.1977 \\
& 0 & 0 & 0 & -14.5784 & 3.7796
\end{bmatrix}
\end{align*}
\]

returns a valid QR factorization, although possibly different from

\[
\begin{align*}
A2 &= A; \\
A2(j,:) &= []; \\
[Q2,R2] &= \text{qr}(A2)
\end{align*}
\]
Algorithm

The qrdelete function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

See Also

planerot, qr, qrinsert
Purpose
Insert column or row into QR factorization

Syntax

\[
\begin{align*}
[Q_1, R_1] &= \text{qrinsert}(Q, R, j, x) \\
[Q_1, R_1] &= \text{qrinsert}(Q, R, j, x, 'col') \\
[Q_1, R_1] &= \text{qrinsert}(Q, R, j, x, 'row')
\end{align*}
\]

Description

\[
[Q_1, R_1] = \text{qrinsert}(Q, R, j, x)
\]
returns the QR factorization of the matrix \( A_1 \), where \( A_1 = QR \) with the column \( x \) inserted before \( A(:, j) \). If \( A \) has \( n \) columns and \( j = n+1 \), then \( x \) is inserted after the last column of \( A \).

\[
[Q_1, R_1] = \text{qrinsert}(Q, R, j, x, 'col')
\]
is the same as
\[
\text{qrinsert}(Q, R, j, x).
\]

\[
[Q_1, R_1] = \text{qrinsert}(Q, R, j, x, 'row')
\]
returns the QR factorization of the matrix \( A_1 \), where \( A_1 = QR \) with an extra row, \( x \), inserted before \( A(j,:) \).

Examples

\[
A = \text{magic}(5);
\]
\[
[Q, R] = \text{qr}(A);
\]
\[
j = 3;
\]
\[
x = 1:5;
\]
\[
[Q_1, R_1] = \text{qrinsert}(Q, R, j, x, 'row')
\]

\[
Q_1 =
\begin{bmatrix}
0.5231 & 0.5039 & -0.6750 & 0.1205 & 0.0411 & 0.0225 \\
0.7078 & -0.6966 & 0.0190 & -0.0788 & 0.0833 & -0.0150 \\
0.0308 & 0.0592 & 0.0656 & 0.1169 & 0.1527 & -0.9769 \\
0.1231 & 0.1363 & 0.3542 & 0.6222 & 0.6398 & 0.2104 \\
0.3077 & 0.1902 & 0.4100 & 0.4161 & -0.7264 & -0.0150 \\
0.3385 & 0.4500 & 0.4961 & -0.6366 & 0.1761 & 0.0225
\end{bmatrix}
\]

\[
R_1 =
\begin{bmatrix}
0 & 19.9292 & 12.4403 & 2.1340 & 4.3271 \\
0 & 0 & 24.4514 & 11.8132 & 3.9931 \\
0 & 0 & 0 & 20.2382 & 10.3392
\end{bmatrix}
\]
returns a valid QR factorization, although possibly different from

\[
A2 = [A(1:j-1,:); x; A(j:end,:)];
\]
\[
[Q2,R2] = qr(A2)
\]

\[
Q2 = \begin{bmatrix}
-0.5231 & 0.5039 & 0.6750 & -0.1205 & 0.0411 & 0.0225 \\
-0.7078 & -0.6966 & -0.0190 & 0.0788 & 0.0833 & -0.0150 \\
-0.0308 & 0.0592 & -0.0656 & -0.1169 & 0.1527 & -0.9769 \\
-0.1231 & 0.1363 & -0.3542 & -0.6222 & 0.6398 & 0.2104 \\
-0.3077 & 0.1902 & -0.4100 & -0.4161 & -0.7264 & -0.0150 \\
-0.3385 & 0.4500 & -0.4961 & 0.6366 & 0.1761 & 0.0225 \\
\end{bmatrix}
\]

\[
R2 = \begin{bmatrix}
0 & 19.9292 & 12.4403 & 2.1340 & 4.3271 \\
0 & 0 & -24.4514 & -11.8132 & -3.9931 \\
0 & 0 & 0 & -20.2382 & -10.3392 \\
0 & 0 & 0 & 0 & 16.1948 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

**Algorithm**

The qrinsert function inserts the values of x into the jth column (row) of R. It then uses a series of Givens rotations to zero out the nonzero elements of R on and below the diagonal in the jth column (row).

**See Also**

planerot, qr, qrdelete
**Description**  Rank 1 update to QR factorization

**Syntax**  
\[ [Q1,R1] = qrupdate(Q,R,u,v) \]

**Description**  
\[ [Q1,R1] = qrupdate(Q,R,u,v) \] when \([Q,R] = qr(A)\) is the original QR factorization of \(A\), returns the QR factorization of \(A + u*v'\), where \(u\) and \(v\) are column vectors of appropriate lengths.

**Remarks**  qrupdate works only for full matrices.

**Examples**  
The matrix
\[
mu = \sqrt{\text{eps}}
\]
\[
mu =
1.4901e-08
\]
\[
A = [\text{ones}(1,4); \text{mu*eye}(4)];
\]
is a well-known example in least squares that indicates the dangers of forming \(A'*A\). Instead, we work with the QR factorization – orthonormal \(Q\) and upper triangular \(R\).
\[
[Q,R] = qr(A);
\]
As we expect, \(R\) is upper triangular.
\[
R =
\begin{bmatrix}
-1.0000 & -1.0000 & -1.0000 & -1.0000 \\
0 & 0.0000 & 0.0000 & 0.0000 \\
0 & 0 & 0.0000 & 0.0000 \\
0 & 0 & 0 & 0.0000 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]
In this case, the upper triangular entries of $R$, excluding the first row, are on the order of $\sqrt{\text{eps}}$.

Consider the update vectors
\[
  u = [-1 0 0 0 0]' \quad \text{and} \quad v = \text{ones}(4,1);
\]

Instead of computing the rather trivial QR factorization of this rank one update to $A$ from scratch with
\[
  [QT,RT] = \text{qr}(A + u*v')
\]

we may use qrupdate.
\[
  [Q1,R1] = \text{qrupdate}(Q,R,u,v)
\]
qrupdate

\[
\begin{array}{ccccc}
0.0000 & 1.0000 & -0.0000 & -0.0000 & 0.0000 \\
0.0000 & 0.0000 & 1.0000 & -0.0000 & 0.0000 \\
-0.0000 & -0.0000 & -0.0000 & 1.0000 & 0.0000 \\
\end{array}
\]

\[R1 = \]

\[
1.0e-007 * \\
0.1490 & 0.0000 & 0.0000 & 0.0000 \\
0 & 0.1490 & 0.0000 & 0.0000 \\
0 & 0 & 0.1490 & 0.0000 \\
0 & 0 & 0 & 0.1490 \\
0 & 0 & 0 & 0 \\
\]

Note that both factorizations are correct, even though they are different.

**Algorithm**

`qrupdate` uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. `qrupdate` is useful since, if we take \( N = \max(m,n) \), then computing the new QR factorization from scratch is roughly an \( O(N^3) \) algorithm, while simply updating the existing factors in this way is an \( O(N^2) \) algorithm.

**References**


**See Also**

`cholupdate`, `qr`
**Purpose**
Numerically evaluate integral, adaptive Simpson quadrature

**Syntax**

\[
\begin{align*}
q &= \text{quad}(\text{fun}, a, b) \\
q &= \text{quad}(\text{fun}, a, b, \text{tol}) \\
q &= \text{quad}(\text{fun}, a, b, \text{tol}, \text{trace}) \\
[q, \text{fcnt}] &= \text{quad}(\ldots)
\end{align*}
\]

**Description**

*Quadrature* is a numerical method used to find the area under the graph of a function, that is, to compute a definite integral.

\[
q = \int_{a}^{b} f(x) \, dx
\]

`q = quad(fun,a,b)` tries to approximate the integral of function `fun` from *a* to *b* to within an error of `1e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. Limits *a* and *b* must be finite. The function \( y = \text{fun}(x) \) should accept a vector argument `x` and return a vector result `y`, the integrand evaluated at each element of `x`.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

`q = quad(fun,a,b,tol)` uses an absolute error tolerance `tol` instead of the default which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results. In MATLAB version 5.3 and earlier, the `quad` function used a less reliable algorithm and a default relative tolerance of `1.0e-3`.

`q = quad(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q]` during the recursion.

`[q,fcnt] = quad(\ldots)` returns the number of function evaluations.

The function `quadl` may be more efficient with high accuracies and smooth integrands.
The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is finite, \([a, \text{Inf})\), then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like \(\log |x - c|\) or \(|x - c|^p\) for \(p \geq -1/2\). If the function is singular at points inside \((a, b)\), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

**Example**

To compute the integral

\[
\int_0^2 \frac{1}{x^3 - 2x - 5} \, dx
\]

write an M-file function `myfun` that computes the integrand:

```matlab
function y = myfun(x)
    y = 1./(x.^3-2*x-5);```

```
Then pass @myfun, a function handle to myfun, to quad, along with the limits of integration, 0 to 2:

\[ Q = \text{quad}(@\text{myfun},0,2) \]

\[ Q = -0.4605 \]

Alternatively, you can pass the integrand to quad as an anonymous function handle F:

\[ F = @(x)1./(x.^3-2*x-5); \]
\[ Q = \text{quad}(F,0,2); \]

**Algorithm**
quad implements a low order method using an adaptive recursive Simpson’s rule.

**Diagnostics**
quad may issue one of the following warnings:

- 'Minimum step size reached' indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

- 'Maximum function count exceeded' indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

- 'Infinite or Not-a-Number function value encountered' indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

**See Also**
quad2d, dblquad, quadgk, quadl, quadv, trapz, triplequad, function_handle (@), “Anonymous Functions”

**References**
**Purpose**
Numerically evaluate double integral over planar region

**Syntax**

```matlab
q = quad2d(fun,a,b,c,d)
[q,errbnd] = quad2d(...)  
q = quad2d(fun,a,b,c,d,param1,val1,param2,val2,...)
```

**Description**

$q = \text{quad2d}(\text{fun},a,b,c,d)$ approximates the integral of $\text{fun}(x,y)$ over the planar region $a \leq x \leq b$ and $c(x) \leq y \leq d(x)$. $\text{fun}$ is a function handle, $c$ and $d$ may each be a scalar or a function handle.

All input functions must be vectorized. The function $Z = \text{fun}(X,Y)$ must accept 2-D matrices $X$ and $Y$ of the same size and return a matrix $Z$ of corresponding values. The functions $\text{ymin}=c(X)$ and $\text{ymax}=d(X)$ must accept matrices and return matrices of the same size with corresponding values.

$[q,\text{errbnd}] = \text{quad2d}(...)$. $\text{errbnd}$ is an approximate upper bound on the absolute error, $|Q - I|$, where $I$ denotes the exact value of the integral.

$q = \text{quad2d}(\text{fun},a,b,c,d,\text{param1},\text{val1},\text{param2},\text{val2},...)$ performs the integration as above with specified values of optional parameters:

<table>
<thead>
<tr>
<th>AbsTol</th>
<th>absolute error tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelTol</td>
<td>relative error tolerance</td>
</tr>
</tbody>
</table>

$\text{quad2d}$ attempts to satisfy $\text{ERRBND} \leq \max(\text{AbsTol},\text{RelTol} \times |Q|)$. This is absolute error control when $|Q|$ is sufficiently small and relative error control when $|Q|$ is larger. A default tolerance value is used when a tolerance is not specified. The default value of $\text{AbsTol}$ is $1e-5$. The default value of $\text{RelTol}$ is $100 \times \text{eps(class}(Q))$. This is also the minimum value of $\text{RelTol}$. Smaller $\text{RelTol}$ values are automatically increased to the default value.

| MaxFunEvals | Maximum allowed number of evaluations of $\text{fun}$ reached. |
The MaxFunEvals parameter limits the number of vectorized calls to fun. The default is 2000.

<table>
<thead>
<tr>
<th>FailurePlot</th>
<th>Generate a plot if MaxFunEvals is reached.</th>
</tr>
</thead>
</table>

Setting FailurePlot to true generates a graphical representation of the regions needing further refinement when MaxFunEvals is reached. No plot is generated if the integration succeeds before reaching MaxFunEvals. These (generally) 4-sided regions are mapped to rectangles internally. Clusters of small regions indicate the areas of difficulty. The default is false.

<table>
<thead>
<tr>
<th>Singular</th>
<th>Problem may have boundary singularities</th>
</tr>
</thead>
</table>

With Singular set to true, quad2d will employ transformations to weaken boundary singularities for better performance. The default is true. Setting Singular to false will turn these transformations off, which may provide a performance benefit on some smooth problems.

**Examples**

**Example 1**

Integrate \( y \sin(x) + x \cos(y) \) over \( \pi \leq x \leq 2\pi, 0 \leq y \leq \pi \). The true value of the integral is \(-\pi^2\).

\[
Q = \text{quad2d}(@(x,y) y.*\sin(x)+x.*\cos(y),\text{pi},2*\text{pi},0,\text{pi})
\]

**Example 2**

Integrate \( [(x+y)^{1/2}(1+x+y)^2]^{-1} \) over the triangle \( 0 \leq x \leq 1 \) and \( 0 \leq y \leq 1-x \). The integrand is infinite at (0,0). The true value of the integral is \( \pi/4-1/2 \).

\[
\text{fun} = @(x,y) 1/((\text{sqrt}(x + y) .* (1 + x + y)).^2 )
\]

In Cartesian coordinates:

\[
\text{ymax} = @(x) 1 - x;
\]
\[ Q = \text{quad2d}(\text{fun}, 0, 1, 0, \text{ymax}) \]

In polar coordinates:

\[
\text{polarfun} = @(\theta, r) \text{fun}(r \cdot \cos(\theta), r \cdot \sin(\theta)) \cdot r; \\
\text{rmax} = @(\theta) 1 / (\sin(\theta) + \cos(\theta)); \\
Q = \text{quad2d}(\text{polarfun}, 0, \pi/2, 0, \text{rmax})
\]

**Limitations**

quad2d begins by mapping the region of integration to a rectangle. Consequently, it may have trouble integrating over a region that does not have four sides or has a side that cannot be mapped smoothly to a straight line. If the integration is unsuccessful, some helpful tactics are leaving \textbf{Singular} set to its default value of \texttt{true}, changing between Cartesian and polar coordinates, or breaking the region of integration into pieces and adding the results of integration over the pieces.

For example:

\[
\text{fun} = @(x, y) \text{abs}(x^2 + y^2 - 0.25); \\
\text{c} = @(x) -\text{sqrt}(1 - x^2); \\
\text{d} = @(x) \text{sqrt}(1 - x^2); \\
\text{quad2d}(\text{fun}, -1, 1, \text{c}, \text{d}, '\text{AbsTol}', 1e-8, ... \\
'\text{FailurePlot}', \text{true}, '\text{Singular}', \text{false})
\]

Warning: Reached the maximum number of function evaluations (2000). The result fails the global error test.

The failure plot shows two areas of difficulty, near the points (-1, 0) and (1, 0) and near the circle \( x^2 + y^2 = 0.25 \):
Changing the value of `Singular` to `true` will cope with the geometric singularities at (-1,0) and (1,0). The larger shaded areas may need refinement but are probably not areas of difficulty.

```matlab
Q = quad2d(fun,-1,1,c,d,'AbsTol',1e-8, ...  
    'FailurePlot',true,'Singular',true)
```

Warning: Reached the maximum number of function evaluations (2000). The result passes the ...

```matlab
global error test.
```
From here you can take advantage of symmetry:

\[
Q = 4 \times \text{quad2d}(\text{fun}, 0, 1, 0, d, \text{Abstol}', 1e-8, \ldots \\
\quad \text{\'Singular', true, \'FailurePlot', true})
\]

However, the code is still working very hard near the singularity. It may not be able to provide higher accuracy:

\[
Q = 4 \times \text{quad2d}(\text{fun}, 0, 1, 0, d, \text{Abstol}', 1e-10, \ldots \\
\quad \text{\'Singular', true, \'FailurePlot', true})
\]

Warning: Reached the maximum number of function ... evaluations (2000). The result passes the ... global error test.
At higher accuracy, a change in coordinates may work better.

```matlab
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
Q = 4*quad2d(polarfun,0,pi/2,0,1,'AbsTol',1e-10)
```

It is best to put the singularity on the boundary by splitting the region of integration into two parts:

```matlab
Q1 = 4*quad2d(polarfun,0,pi/2,0,0.5,'AbsTol',5e-11);
Q2 = 4*quad2d(polarfun,0,pi/2,0.5,1,'AbsTol',5e-11);
Q = Q1 + Q2
```

References


See Also
dblquad, quad, quadl, quadv, quadgk, triplequad, function_handle (@), “Anonymous Functions”
Purpose

Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Syntax

q = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b,tol)
[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)

Description

q = quadgk(fun,a,b) attempts to approximate the integral of a scalar-valued function fun from a to b using high-order global adaptive quadrature and default error tolerances. The function y = fun(x) should accept a vector argument x and return a vector result y. The integrand evaluated at each element of x. fun must be a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. Limits a and b can be -Inf or Inf. If both are finite, they can be complex. If at least one is complex, the integral is approximated over a straight line path from a to b in the complex plane.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function fun, if necessary.

[q,errbnd] = quadgk(fun,a,b,tol) returns an approximate bound on the absolute error, |Q - I|, where I denotes the exact value of the integral.

[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...) performs the integration with specified values of optional parameters. The available parameters are
| Parameter | Description | quadgk attempts to satisfy errbnd <= max(AbsTol,RelTol*|Q|) |
|-----------|-------------|-----------------------------------------------------------|
| 'AbsTol'  | Absolute error tolerance. The default value of 'AbsTol' is 1.e-10 (double), 1.e-5 (single). | This is absolute error control when |Q| is sufficiently small and relative error control when |Q| is larger. For pure absolute error control use 'AbsTol' > 0 and 'RelTol' = 0. For pure relative error control use 'AbsTol' = 0. Except when using pure absolute error control, the minimum relative tolerance is 'RelTol' >= 100*eps(class(Q)). |
| 'RelTol'  | Relative error tolerance. The default value of 'RelTol' is 1.e-6 (double), 1.e-4 (single). | |

quadgk
### quadgk

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| 'Waypoints'    | Vector of integration waypoints. If \( \text{fun}(x) \) has discontinuities in the interval of integration, the locations should be supplied as a Waypoints vector. When \( a, b, \) and the waypoints are all real, only the waypoints between \( a \) and \( b \) are used, and they are used in sorted order. Note that waypoints are not intended for singularities in \( \text{fun}(x) \). Singular points should be handled by making them endpoints of separate integrations and adding the results.

If \( a, b, \) or any entry of the waypoints vector is complex, the integration is performed over a sequence of straight line paths in the complex plane, from \( a \) to the first waypoint, from the first waypoint to the second, and so forth, and finally from the last waypoint to \( b \).

| 'MaxIntervalCount' | Maximum number of intervals allowed. The default value is 650. The 'MaxIntervalCount' parameter limits the number of intervals that quadgk uses at any one time after the first iteration. A warning is issued if quadgk returns early because of this limit. Routinely increasing this value is not recommended, but it can be necessary. |
The list below contains information to help you determine which quadrature function in MATLAB to use:

- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quadl function may be more efficient than quad at higher accuracies with smooth integrands.
- The quadgk function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, \([a, \infty)\), then for the integral of \(\text{fun}(x)\) to exist, \(\text{fun}(x)\) must decay as \(x\) approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if \(\text{fun}(x)\) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint \(c\) like \(\log |x-c|\) or \(|x-c|^{p}\) for \(p \geq -1/2\). If the function is singular at points inside \((a,b)\), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

**Examples**

**Integrand with a singularity at an integration end point**

Write an M-file function `myfun` that computes the integrand:

```matlab
function y = myfun(x)
    y = exp(x).*log(x);
```

Then pass `@myfun`, a function handle to `myfun`, to `quadgk`, along with the limits of integration, 0 to 1:
Q = quadgk(@myfun,0,1)

Q =

-1.3179

Alternatively, you can pass the integrand to `quadgk` as an anonymous function handle `F`:

```matlab
F = (@(x)exp(x).*log(x));
Q = quadgk(F,0,1);
```

**Oscillatory integrand on a semi-infinite interval**

Integrate over a semi-infinite interval with specified tolerances, and return the approximate error bound:

```matlab
[q,errbnd] = quadgk(@(x)x.^5.*exp(-x).*sin(x),0,inf,'RelTol',1e-8,'AbsTol',1e-9);
```

q =

-15.0000

errbnd =

9.4386e-009

**Contour integration around a pole**

Use `Waypoints` to integrate around a pole using a piecewise linear contour:

```matlab
Q = quadgk(@(z)1./(2*z - 1),-1-i,-1-i,'Waypoints',[1-i,1+i,-1+i])
```

Q =

0.0000 + 3.1416i
Algorithm
quadgk implements adaptive quadrature based on a Gauss-Kronrod pair (15th and 7th order formulas).

Diagnostics
quadgk may issue one of the following warnings:

'Minimum step size reached' indicates that interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

'Reached the limit on the maximum number of intervals in use' indicates that the integration was terminated before meeting the tolerance requirements and that continuing the integration would require more than MaxIntervalCount subintervals. The integral may not exist, or it may be difficult to approximate numerically. Increasing MaxIntervalCount usually does not help unless the tolerance requirements were nearly met when the integration was previously terminated.

'Infinite or Not-a-Number function value encountered' indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

References

See Also
quad2d, db1quad, quad, quad1, quadl, quadv, triplequad, function_handle (@), “Anonymous Functions”
Purpose
Numerically evaluate integral, adaptive Lobatto quadrature

Syntax
q = quadl(fun,a,b)
q = quadl(fun,a,b,tol)
quadl(fun,a,b,tol,trace)
[q,fcnt] = quadl(...)  

Description
q = quadl(fun,a,b) approximates the integral of function fun from a to b, to within an error of $10^{-6}$ using recursive adaptive Lobatto quadrature. fun is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. fun accepts a vector x and returns a vector y, the function fun evaluated at each element of x. Limits a and b must be finite.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function fun, if necessary.

q = quadl(fun,a,b,tol) uses an absolute error tolerance of tol instead of the default, which is $1.0e-6$. Larger values of tol result in fewer function evaluations and faster computation, but less accurate results.

quadl(fun,a,b,tol,trace) with non-zero trace shows the values of [fcnt a b-a q] during the recursion.

[q,fcnt] = quadl(...) returns the number of function evaluations.

Use array operators .*, ./ and .^ in the definition of fun so that it can be evaluated with a vector argument.

The function quad may be more efficient with low accuracies or nonsmooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

• The quad function may be most efficient for low accuracies with nonsmooth integrands.
The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.

The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.

The `quadv` function vectorizes `quad` for an array-valued `fun`.

If the interval is infinite, \([a, \infty)\), then for the integral of \(f(x)\) to exist, \(f(x)\) must decay as \(x\) approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if \(f(x)\) decays fast enough.

The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint \(c\) like \(\log|x-c|\) or \(|x-c|^{p}\) for \(p \geq -1/2\). If the function is singular at points inside \((a,b)\), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

### Examples

Pass M-file function handle `@myfun` to `quadl`:

```matlab
Q = quadl(@myfun,0,2);
```

where the M-file `myfun.m` is

```matlab
function y = myfun(x)
    y = 1./(x.^3-2*x-5);
```

Pass anonymous function handle `F` to `quadl`:

```matlab
F = @(x) 1./(x.^3-2*x-5);
Q = quadl(F,0,2);
```

### Algorithm

`quadl` implements a high order method using an adaptive Gauss/Lobatto quadrature rule.
quadl

Diagnostics
quadl may issue one of the following warnings:

'Minimum step size reached' indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

'Maximum function count exceeded' indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

'Infinite or Not-a-Number function value encountered' indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

See Also
quad2d, dblquad, quad, quadgk, triplequad, function_handle (@), “Anonymous Functions”

References
Purpose
Vectorized quadrature

Syntax
Q = quadv(fun,a,b)
Q = quadv(fun,a,b,tol)
Q = quadv(fun,a,b,tol,trace)
[Q,fcnt] = quadv(…)

Description
Q = quadv(fun,a,b) approximates the integral of the complex array-valued function fun from a to b within an error of 1.e-6 using recursive adaptive Simpson quadrature. fun is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. The function \( Y = \text{fun}(x) \) should accept a scalar argument x and return an array result Y, whose components are the integrands evaluated at x. Limits a and b must be finite.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide addition parameters to the function fun, if necessary.

Q = quadv(fun,a,b,tol) uses the absolute error tolerance tol for all the integrals instead of the default, which is 1.e-6.

Note The same tolerance is used for all components, so the results obtained with quadv are usually not the same as those obtained with quad on the individual components.

Q = quadv(fun,a,b,tol,trace) with non-zero trace shows the values of [fcnt a b-a Q(1)] during the recursion.

[Q,fcnt] = quadv(…) returns the number of function evaluations.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The quad function may be most efficient for low accuracies with nonsmooth integrands.
• The `quad1` function may be more efficient than `quad` at higher accuracies with smooth integrands.

• The `quadgk` function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.

• The `quadv` function vectorizes `quad` for an array-valued `fun`.

• If the interval is infinite, `[a, Inf]`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.

• The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log |x-c|` or `|x-c|^p` for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

**Example**

For the parameterized array-valued function `myarrayfun`, defined by

```matlab
function Y = myarrayfun(x,n)
    Y = 1 ./ ((1:n)+x);
```

the following command integrates `myarrayfun`, for the parameter value `n = 10` between `a = 0` and `b = 1`:

```matlab
Qv = quadv(@(x)myarrayfun(x,10),0,1);
```

The resulting array `Qv` has 10 elements estimating `Q(k) = log((k+1)./(k))`, for `k = 1:10`.

The entries in `Qv` are slightly different than if you compute the integrals using `quad` in a loop:
for k = 1:10
    Qs(k) = quadv(@(x)myscalarfun(x,k),0,1);
end

where myscalarfun is:

function y = myscalarfun(x,k)
    y = 1./(k+x);

See Also
quad, quad2d, quadgk, quad1, dblquad, triplequad, function_handle
Purpose
Create and open question dialog box

Syntax
button = questdlg('qstring')
button = questdlg('qstring','title')
button = questdlg('qstring','title',default)
button = questdlg('qstring','title','str1','str2',default)
button = questdlg('qstring','title','str1','str2','str3', default)
button = questdlg('qstring','title', ... , options)

Description
button = questdlg('qstring') displays a modal dialog box presenting the question 'qstring'. The dialog has three default buttons, Yes, No, and Cancel. If the user presses one of these three buttons, button is set to the name of the button pressed. If the user presses the close button on the dialog without making a choice, button is set to the empty string. If the user presses the Return key, button is set to 'Yes'. 'qstring' is a cell array or a string that automatically wraps to fit within the dialog box.

Note
A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in the MATLAB Figure Properties.

button = questdlg('qstring','title') displays a question dialog with 'title' displayed in the dialog's title bar.

button = questdlg('qstring','title',default) specifies which push button is the default in the event that the Return key is pressed. 'default' must be 'Yes', 'No', or 'Cancel'.

button = questdlg('qstring','title','str1','str2',default) creates a question dialog box with two push buttons labeled 'str1' and 'str2'. default specifies the default button selection and must be 'str1' or 'str2'.

2-2958
button = questdlg('qstring','title','str1','str2','str3','default') creates a question dialog box with three push buttons labeled 'str1', 'str2', and 'str3'. default specifies the default button selection and must be 'str1', 'str2', or 'str3'.

When default is specified, but is not set to one of the button names, pressing the Enter key displays a warning and the dialog remains open.

button = questdlg('qstring','title', ..., options) replaces the string default with a structure, options. The structure specifies which button string is the default answer, and whether to use TeX to interpret the question string, qstring. Button strings and dialog titles cannot use TeX interpretation. The options structure must include the fields Default and Interpreter, both strings. It can include other fields, but questdlg does not use them. You can set Interpreter to 'none' or 'tex'. If the Default field does not contain a valid button name, a command window warning is issued and the dialog box does not respond to pressing the Enter key.

Examples

Example 1

Create a dialog that requests a dessert preference and encode the resulting choice as an integer.

% Construct a questdlg with three options
choice = questdlg('Please choose a dessert:', ...
    'Dessert Menu', ...
    'Ice cream','Cake','No thank you','No thank you');
% Handle response
switch choice
    case 'Ice cream'
        disp([choice ' coming right up.'])
        dessert = 1;
        break
    case 'Cake'
        disp([choice ' coming right up.'])
        dessert = 2;
        break
    % Add more cases for other options...
end
case 'No thank you'
    disp('I''ll bring you your check.')
    dessert = 0;
end

The case statements can contain white space but are case-sensitive.

Example 2

Specify an options structure to use the TeX interpreter to format a question.

    options.Interpreter = 'tex';
    % Include the desired Default answer
    options.Default = 'Don''t know';
    % Create a TeX string for the question
    qstring = 'Is \Sigma(\alpha - \beta) < 0?';
    choice = questdlg(qstring,'Boundary Condition',...
    'Yes','No','Don''t know',options)
See Also

dialog, errordlg, helpdlg, inputdlg, listdlg, msgbox, warndlg
figure, textwrap, uwait, uiresume

Predefined Dialog Boxes for related functions
Purpose

Terminate MATLAB program

GUI Alternatives

As an alternative to the quit function, use the Close box or select File > Exit MATLAB in the MATLAB desktop.

Syntax

```
quit
quit cancel
quit force
```

Description

quit displays a confirmation dialog box if the confirm upon quitting preference is selected, and if confirmed or if the confirmation preference is not selected, terminates MATLAB after running finish.m, if finish.m exists. The workspace is not automatically saved by quit. To save the workspace or perform other actions when quitting, create a finish.m file to perform those actions. For example, you can display a custom dialog box to confirm quitting using a finish.m file—see the following examples for details. If an error occurs while finish.m is running, quit is canceled so that you can correct your finish.m file without losing your workspace.

```
quit cancel
```

is for use in finish.m and cancels quitting. It has no effect anywhere else.

```
quit force
```

bypasses finish.m and terminates MATLAB. Use this to override finish.m, for example, if an errant finish.m will not let you quit.

Remarks

When using Handle Graphics objects in finish.m, use uiswait, waitfor, or drawnow so that figures are visible. See the reference pages for these functions for more information.

If you want MATLAB to display the following confirmation dialog box after running quit, select File > Preferences > General > Confirmation Dialogs. Then select the check box for Confirm before exiting MATLAB, and click OK.
Examples

Two sample `finish.m` files are included with MATLAB. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finish sav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finish dlg.m`—Displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code:

```matlab
button = questdlg('Ready to quit?', ...    
   'Exit Dialog','Yes','No','No');    
switch button
   case 'Yes',
      disp('Exiting MATLAB');        
      %Save variables to matlab.mat
      save
   case 'No',
      quit cancel;
end
```

See Also

`exit`, `finish`, `save`, `startup`
**Quit (COM)**

**Purpose**
Terminate MATLAB Automation server

**Syntax**

**MATLAB Client**
- `h.Quit`
- `Quit(h)`
- `invoke(h, 'Quit')`

**Method Signature**
- `void Quit(void)`

**Microsoft Visual Basic Client**
- `Quit`

**Description**
Quit terminates the MATLAB server session attached to handle `h`.

**Remarks**
Server function names, like `Quit`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

COM functions are available on Microsoft Windows systems only.
**Purpose**

Quiver or velocity plot

**GUI Alternatives**

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

quiver(x,y,u,v)
quiver(u,v)
quiver(...,scale)
quiver(...,LineSpec)
quiver(...,LineSpec,'filled')
quiver(axes_handle,...)
h = quiver(...)
hlines = quiver('v6',...)

**Description**

A quiver plot displays velocity vectors as arrows with components \((u,v)\) at the points \((x,y)\).

For example, the first vector is defined by components \(u(1), v(1)\) and is displayed at the point \(x(1), y(1)\).

\[
quiver(x,y,u,v)
\]

plots vectors as arrows at the coordinates specified in each corresponding pair of elements in \(x\) and \(y\). The matrices \(x, y, u,\) and \(v\) must all be the same size and contain corresponding position and velocity components. However, \(x\) and \(y\) can also be vectors, as explained in the next section. By default, the arrows are scaled to just not overlap, but you can scale them to be longer or shorter if you want.

**Expanding x- and y-Coordinates**

MATLAB expands \(x\) and \(y\) if they are not matrices. This expansion is equivalent to calling *meshgrid* to generate matrices from vectors:
[x,y] = meshgrid(x,y);
quiver(x,y,u,v)

In this case, the following must be true:
length(x) = n and length(y) = m, where [m,n] = size(u) = size(v).
The vector x corresponds to the columns of u and v, and vector y corresponds to the rows of u and v.
quiver(u,v) draws vectors specified by u and v at equally spaced points in the x-y plane.
quiver(...,scale) automatically scales the arrows to fit within the grid and then stretches them by the factor scale. scale = 2 doubles their relative length, and scale = 0.5 halves the length. Use scale = 0 to plot the velocity vectors without automatic scaling. You can also tune the length of arrows after they have been drawn by choosing the Plot Edit tool, selecting the quivergroup object, opening the Property Editor, and adjusting the Length slider.
quiver(...,LineSpec) specifies line style, marker symbol, and color using any valid LineSpec. quiver draws the markers at the origin of the vectors.
quiver(...,LineSpec,'filled') fills markers specified by LineSpec.
quiver(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes (gca).
h = quiver(...) returns the handle to the quivergroup object.

**Backward-Compatible Version**
hlines = quiver('v6',...) returns the handles of line objects instead of quivergroup objects for compatibility with MATLAB 6.5 and earlier.
Note The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Examples

**Showing the Gradient with Quiver Plots**

Plot the gradient field of the function $Z = xe^{-x^2 - y^2}$.

```matlab
[X,Y] = meshgrid(-2:.2:2);
Z = X.*exp(-X.^2 - Y.^2);
[DX, DY] = gradient(Z, .2, .2);
contour(X, Y, Z)
hold on
quiver(X, Y, DX, DY)
colormap hsv
hold off
```
**See Also**

- `contour`, `LineSpec`, `plot`, `quiver3`
- “Direction and Velocity Plots” on page 1-96 for related functions
- Two-Dimensional Quiver Plots for more examples
- `Quivergroup Properties` for property descriptions
Purpose

3-D quiver or velocity plot

![Quiver3 diagram](image)

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

```matlab
quiver3(x,y,z,u,v,w)
quiver3(z,u,v,w)
quiver3(...,scale)
quiver3(...,LineSpec)
quiver3(...,LineSpec,'filled')
quiver3(axes_handle,...)
h = quiver3(...)
```

Description

A three-dimensional quiver plot displays vectors with components \((u,v,w)\) at the points \((x,y,z)\), where \(u,v,w,x,y,\) and \(z\) all have real (non-complex) values.

\(\text{quiver3}(x,y,z,u,v,w)\) plots vectors with components \((u,v,w)\) at the points \((x,y,z)\). The matrices \(x,y,z,u,v,w\) must all be the same size and contain the corresponding position and vector components.

\(\text{quiver3}(z,u,v,w)\) plots the vectors at the equally spaced surface points specified by matrix \(z\). \(\text{quiver3}\) automatically scales the vectors based on the distance between them to prevent them from overlapping.

\(\text{quiver3}(...,\text{scale})\) automatically scales the vectors to prevent them from overlapping, and then multiplies them by \(\text{scale}\). \(\text{scale} = 2\) doubles their relative length, and \(\text{scale} = 0.5\) halves them. Use \(\text{scale} = 0\) to plot the vectors without the automatic scaling.
quiver3(...) specifies line type and color using any valid LineSpec.

quiver3(...,LineSpec,'filled') fills markers specified by LineSpec.

quiver3(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes (gca).

h = quiver3(...) returns a vector of line handles.

Examples

Plot the surface normals of the function $Z = xe^{(-x^2 - y^2)}$.

[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);
quiver3(X,Y,Z,U,V,W,0.5);
hold on
surf(X,Y,Z);
colormap hsv
view(-35,45)
axis ([ -2 2 -1 1 -.6 .6])
hold off
See Also

axis, contour, LineSpec, plot, plot3, quiver, surfnorm, view

“Direction and Velocity Plots” on page 1-96 for related functions

Three-Dimensional Quiver Plots for more examples
### Quivergroup Properties

**Purpose**
Define quivergroup properties

**Modifying Properties**
You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default properties for areaseries objects.

See Plot Objects for more information on quivergroup objects.

**Quivergroup Property Descriptions**
This section provides a description of properties. Curly braces `{}` enclose default values.

**Annotation**

hg.Annotation object Read Only

*Control the display of quivergroup objects in legends.* The Annotation property enables you to specify whether this quivergroup object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the quivergroup object is displayed in a figure legend:

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Include the quivergroup object in a legend as one entry, but not its children objects</td>
</tr>
<tr>
<td>off</td>
<td>Do not include the quivergroup or its children in a legend (default)</td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the quivergroup as separate entries in the legend</td>
</tr>
</tbody>
</table>
Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle hobj to children, which causes each child object to have an entry in the legend:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

AutoScale
{on} | off

*Autoscale arrow length.* Based on average spacing in the x and y directions, AutoScale scales the arrow length to fit within the grid-defined coordinate data and keeps the arrows from overlapping. After autoscaling, quiver applies the AutoScaleFactor to the arrow length.

AutoScaleFactor
scalar (default = 0.9)

*User-specified scale factor.* When AutoScale is on, the quiver function applies this user-specified autoscale factor to the arrow length. A value of 2 doubles the length of the arrows; 0.5 halves the length.

BeingDeleted
on | {off} Read Only

*This object is being deleted.* The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object’s delete function callback is called.
Quivergroup Properties

(see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, an object's delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object's BeingDeleted property before acting.

BusyAction
cancel | {queue}

*Callback routine interruption.* The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- cancel — Discard the event that attempted to execute a second callback routine.
- queue — Queue the event that attempted to execute a second callback routine until the current callback finishes.

ButtonDownFcn
string or function handle

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.
See the figure’s `SelectionType` property to determine if modifier keys were also pressed.

This property can be
- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

**Children**
array of graphics object handles

*Children of this object.* The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's `HandleVisibility` property is set to callback or off, its handle does not show up in this object's `Children` property unless you set the root `ShowHiddenHandles` property to on:

```matlab
set(0,'ShowHiddenHandles','on')
```

**Clipping**
{on} | off

*Clipping mode.* MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to off, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set `hold` to on, freeze axis scaling (`axis manual`), and then create a larger plot object.
Quivergroup Properties

**Color**

**ColorSpec**

*Color of the object.* A three-element RGB vector or one of the MATLAB predefined names, specifying the object’s color.

See the **ColorSpec** reference page for more information on specifying color.

**CreateFcn**

string or function handle

*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y, 'CreateFcn', @CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose **CreateFcn** is being executed is accessible only through the root **CallbackObject** property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

**DeleteFcn**

string or function handle

*Callback executed during object deletion.* A callback that executes when this object is deleted (e.g., this might happen when you issue
a delete command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which can be queried using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the BeingDeleted property for related information.

DisplayName
string (default is empty string)

String used by legend for this quivergroup object. The legend function uses the string defined by the DisplayName property to label this quivergroup object in the legend.

- If you specify string arguments with the legend function, DisplayName is set to this quivergroup object’s corresponding string and that string is used for the legend.

- If DisplayName is empty, legend creates a string of the form, ['data' n], where n is the number assigned to the object based on its location in the list of legend entries. However, legend does not set DisplayName to this string.

- If you edit the string directly in an existing legend, DisplayName is set to the edited string.

- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.

- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.
See “Controlling Legends” for more examples.

EraseMode
\{normal\} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with **EraseMode none**, you cannot print these objects because MATLAB stores no information about their former locations.

- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes **Color** property is set to **none**). That is, it isn’t erased correctly if there are objects behind it.

- **background** — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes **Color** property is set to **none**). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

**Printing with Nonnormal Erase Modes**
MATLAB always prints figures as if the `EraseMode` of all objects is normal. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the `Axes` `Color` property. Set the figure background color with the `Figure` `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

**HandleVisibility**

```plaintext
{on} | callback | off
```

*Control access to object’s handle by command-line users and GUIs.* This property determines when an object’s handle is visible in its parent’s list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- **on** — Handles are always visible when `HandleVisibility` is on.
- **callback** — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- **off** — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.
Functions Affected by Handle Visibility

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle’s visibility is restricted using `callback` or `off`, the object’s handle does not appear in its parent’s `Children` property, figures do not appear in the root’s `CurrentFigure` property, objects do not appear in the root’s `CallbackObject` property or in the figure’s `CurrentObject` property, and axes do not appear in their parent’s `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties and pass it to any function that operates on handles.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Quivergroup Properties

HitTest
{on} | off

Selectable by mouse click. HitTest determines whether this object can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click on the objects that compose the area graph. If HitTest is off, clicking this object selects the object below it (which is usually the axes containing it).

HitTestArea
on | {off}

Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When HitTestArea is off, you must click the object’s lines or markers (excluding the baseline, if any) to select the object. When HitTestArea is on, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

Interruptible
{on} | off

Callback routine interruption mode. The Interruptible property controls whether an object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.
Quivergroup Properties

Setting Interruptible to on allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the gca or gcf command) when an interruption occurs.

LineStyle
{-} | – | : | . | none

*Line style.* This property specifies the line style of the object. Available line styles are shown in the following table.

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>--</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

You can use LineStyle none when you want to place a marker at each point but do not want the points connected with a line (see the Marker property).

LineWidth
scalar

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = \(1/72\) inch). The default LineWidth is 0.5 points.

Marker
character (see table)

*Marker symbol.* The Marker property specifies the type of markers that are displayed at plot vertices. You can set values for the
Marker property independently from the LineStyle property. Supported markers include those shown in the following table.

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>o</td>
<td>Circle</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

**MarkerEdgeColor**

ColorSpec | none | {auto}

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the Color property.

**MarkerFaceColor**

ColorSpec | {none} | auto
Quivergroup Properties

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or to the figure color if the axes Color property is set to none (which is the factory default for axes objects).

MarkerSize
size in points

Marker size. A scalar specifying the size of the marker in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the '.' symbol) at one-third the specified size.

MaxHeadSize
scalar (default = 0.2)

Maximum size of arrowhead. A value determining the maximum size of the arrowhead relative to the length of the arrow.

Parent
handle of parent axes, hggroup, or hgtransform

Parent of this object. This property contains the handle of the object’s parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

Selected
on | {off}

Is object selected? When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this
property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

**SelectionHighlight**

{on} | off

*Objects are highlighted when selected.* When the **Selected** property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When **SelectionHighlight** is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

**ShowArrowHead**

{on} | off

*Display arrowheads on vectors.* When this property is on, MATLAB draws arrowheads on the vectors displayed by **quiver**. When you set this property to off, **quiver** draws the vectors as lines without arrowheads.

**Tag**

string

*User-specified object label.* The **Tag** property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define **Tag** as any string.

For example, you might create an area series object and set the **Tag** property.

```matlab
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use **findobj** to find the object’s handle. The following statement changes the **FaceColor** property of the object whose **Tag** is `area1`. 
Quivergroup Properties

```matlab
set(findobj('Tag','area1'),'FaceColor','red')
```

**Type**

string (read only)

*Type of graphics object.* This property contains a string that identifies the class of the graphics object. For stem objects, Type is 'hggroup'. This statement finds all the hggroup objects in the current axes.

```matlab
t = findobj(gca,'Type','hggroup');
```

**UIContextMenu**

handle of a uicontextmenu object

*Associate a context menu with this object.* Assign this property the handle of a uicontextmenu object created in the object’s parent figure. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

**UserData**

array

*User-specified data.* This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

**Visible**

{on} | off

*Visibility of this object and its children.* By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s `Visible` property is set to off. Setting an object’s `Visible` property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.
UData

matrix

One dimension of 2-D or 3-D vector components. UData, VData, and WData, together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components UData(1), VData(1), WData(1).

UDataSource

string (MATLAB variable)

Link UData to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the UData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change UData.

You can use the refreshdata function to force an update of the object's data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

Note If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

VData

matrix
One dimension of 2-D or 3-D vector components. UData, VData and WData (for 3-D) together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components UData(1), VData(1), WData(1).

VDataSource

string (MATLAB variable)

Link VData to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the VData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change VData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

**Note** If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

WData

matrix

One dimension of 2-D or 3-D vector components. UData, VData and WData (for 3-D) together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components UData(1), VData(1), WData(1).
**WDataSource**

string (MATLAB variable)

*Link WData to MATLAB variable.* Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the WData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change WData.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

**Note** If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

**XData**

vector or matrix

*X-axis coordinates of arrows.* The `quiver` function draws an individual arrow at each x-axis location in the XData array. XData can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of columns in UData or VData. That is, `length(XData) == size(UData,2)`.

If you do not specify XData (i.e., the input argument X), the `quiver` function uses the indices of UData to create the quiver graph. See the XDataMode property for related information.
Quivergroup Properties

XDataMode
{auto} | manual

*Use automatic or user-specified x-axis values.* If you specify XData (by setting the XData property or specifying the input argument X), the quiver function sets this property to manual.

If you set XDataMode to auto after having specified XData, the quiver function resets the x tick-mark labels to the indices of the U, V, and W data, overwriting any previous values.

XDataSource
string (MATLAB variable)

*Link XData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change XData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
YData

vector or matrix

*Y-axis coordinates of arrows.* The quiver function draws an individual arrow at each y-axis location in the YData array. YData can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of rows in UData or VData. That is, length(YData) == size(UData,1).

If you do not specify YData (i.e., the input argument Y), the quiver function uses the indices of VData to create the quiver graph. See the YDataMode property for related information.

The input argument y in the quiver function calling syntax assigns values to YData.

YDataMode

{auto} | manual

*Use automatic or user-specified y-axis values.* If you specify YData (by setting the YData property or specifying the input argument Y), MATLAB sets this property to manual.

If you set YDataMode to auto after having specified YData, MATLAB resets the y tick-mark labels to the indices of the U, V, and W data, overwriting any previous values.

YDataSource

string (MATLAB variable)

*Link YData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change YData.
You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

---

**ZData**

vector or matrix

*Z-axis coordinates of arrows.* The quiver function draws an individual arrow at each z-axis location in the ZData array. ZData must be a matrix equal in size to XData and YData.

The input argument z in the `quiver3` function calling syntax assigns values to ZData.
**Purpose**
QZ factorization for generalized eigenvalues

**Syntax**

\[
[AA, BB, Q, Z] = qz(A, B) \\
[AA, BB, Q, Z, V, W] = qz(A, B) \\
qz(A, B, flag)
\]

**Description**
The `qz` function gives access to intermediate results in the computation of generalized eigenvalues.

\[
[AA, BB, Q, Z] = qz(A, B)
\]
for square matrices \(A\) and \(B\), produces upper quasitriangular matrices \(AA\) and \(BB\), and unitary matrices \(Q\) and \(Z\) such that \(Q*A*Z = AA\), and \(Q*B*Z = BB\). For complex matrices, \(AA\) and \(BB\) are triangular.

\[
[AA, BB, Q, Z, V, W] = qz(A, B)
\]
also produces matrices \(V\) and \(W\) whose columns are generalized eigenvectors.

\[
qz(A, B, flag)
\]
for real matrices \(A\) and \(B\), produces one of two decompositions depending on the value of \(flag\):

- **'complex'**
  Produces a possibly complex decomposition with a triangular \(AA\). For compatibility with earlier versions, 'complex' is the default.

- **'real'**
  Produces a real decomposition with a quasitriangular \(AA\), containing 1-by-1 and 2-by-2 blocks on its diagonal.

If \(AA\) is triangular, the diagonal elements of \(AA\) and \(BB\), \(\alpha = \text{diag}(AA)\) and \(\beta = \text{diag}(BB)\), are the generalized eigenvalues that satisfy

\[
A*V*\beta = B*V*\alpha \\
\beta*W'*A = \alpha*W'*B
\]

The eigenvalues produced by

\[
\lambda = \text{eig}(A, B)
\]
are the ratios of the \(\alpha\)s and \(\beta\)s.
If $AA$ is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

**Algorithm**

For full matrices $A$ and $B$, qz uses the LAPACK routines listed in the following table.

<table>
<thead>
<tr>
<th></th>
<th>$A$ and $B$ Real</th>
<th>$A$ or $B$ Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>A and B double</td>
<td>DGGES, DTGEVC (if you request the fifth output $V$)</td>
<td>ZGGES, ZTGEVC (if you request the fifth output $V$)</td>
</tr>
<tr>
<td>A or B single</td>
<td>SGGES, STGEVC (if you request the fifth output $V$)</td>
<td>CGGES, CTGEVC (if you request the fifth output $V$)</td>
</tr>
</tbody>
</table>

**See Also**

eig

**References**

Purpose

Uniformly distributed pseudorandom numbers

Syntax

\[
\begin{align*}
r &= \text{rand}(n) \\
r &= \text{rand}(m,n) \\
r &= \text{rand}([m,n]) \\
r &= \text{rand}(m,n,p,...) \\
r &= \text{rand}([m,n,p,...]) \\
r &= \text{rand} \quad (\text{optional}) \\
r &= \text{rand}(\text{size}(A)) \\
r &= \text{rand}(\text{size}(A), \text{'double'}) \\
r &= \text{rand}(\text{size}(A), \text{'single'})
\end{align*}
\]

Description

\(r = \text{rand}(n)\) returns an \(n\)-by-\(n\) matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval \((0,1)\). \(\text{rand}(m,n)\) or \(\text{rand}([m,n])\) returns an \(m\)-by-\(n\) matrix. \(\text{rand}(m,n,p,...)\) or \(\text{rand}([m,n,p,...])\) returns an \(m\)-by-\(n\)-by-\(p\)-by-... array. \(\text{rand}\) returns a scalar. \(\text{rand}([\text{size}(A)])\) returns an array the same size as \(A\).

\(r = \text{rand}(\text{size}(A), \text{'double'})\) or \(r = \text{rand}(\text{size}(A), \text{'single'})\) returns an array of uniform values of the specified class.

Note

Note: The size inputs \(m, n, p, ...\) should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by \text{rand} is determined by the internal state of the uniform pseudorandom number generator that underlies \text{rand}, \text{randi}, and \text{randn}. The default random number stream properties can be set using \text{@RandStream} methods. See \text{@RandStream} for details about controlling the default stream.

Resetting the default stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties. Since the random number generator is initialized to the same state every time MATLAB software starts up, \text{rand}, \text{randn}, and \text{randi} will
generate the same sequence of numbers in each session until the state is changed.

**Note** In versions of MATLAB prior to 7.7, you controlled the internal state of the random number stream used by rand by calling rand directly with the 'seed', 'state', or 'twister' keywords. That syntax is still supported for backwards compatibility, but is deprecated. For version 7.7, use the default stream as described in the @RandStream reference documentation.

**Examples**

Generate values from the uniform distribution on the interval \([a, b]\).

\[
r = a + (b-a).*\text{rand}(100,1);
\]

Replace the default stream at MATLAB startup, using a stream whose seed is based on clock, so that rand will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session.

```matlab
RandStream.setDefaultStream
    (RandStream('mt19937ar','seed',sum(100*clock)));
rand(1,5)
```

Save the current state of the default stream, generate 5 values, restore the state, and repeat the sequence.

```matlab
defaultStream = RandStream.getDefaultStream;
savedState = defaultStream.State;
u1 = rand(1,5)
defaultStream.State = savedState;
u2 = rand(1,5) % contains exactly the same values as u1
```

**See Also**

randi, randn, @RandStream, rand (RandStream), getDefaultStream (RandStream), sprand, sprandn, randperm
Purpose
Uniformly distributed random numbers

Class
@RandStream

Syntax

```
r = rand(s,n)
rand(s,m,n)
rand(s,[m,n])
rand(s,m,n,p,...)
rand(s,[m,n,p,...])
rand(s)
rand(s,size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

```
r = rand(s,n) returns an n-by-n matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). The values are drawn from the random stream s. rand(s,m,n) or rand(s,[m,n]) returns an m-by-n matrix. rand(s,m,n,p,...) or rand(s,[m,n,p,...]) returns an m-by-n-by-p-by-... array. rand(s) returns a scalar. rand(s,size(A)) returns an array the same size as A.

r = rand(..., 'double') or r = rand(..., 'single') returns an array of uniform values of the specified class.
```

Note
The size inputs m, n, p, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by rand is determined by the internal state of the random number stream s. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.
See Also

rand, @RandStream, randi (RandStream), randn (RandStream),
randperm (RandStream)
**Purpose**
Uniformly distributed pseudorandom integers

**Syntax**
- `randi(imax)`
- `r = randi(imax,n)`
- `randi(imax,m,n)`
- `randi(imax,[m,n])`
- `randi(imax,m,n,p,...)`
- `randi(imax,[m,n,p,...])`
- `randi(imax,size(A))`
- `r = randi([imin,imax],...)`
- `r = randi(..., classname)`

**Description**
`randi(imax)` returns a random integer on the interval 1:imax. 
`r = randi(imax,n)` returns an n-by-n matrix containing pseudorandom integer values drawn from the discrete uniform distribution on 1:imax.
`randi(imax,m,n)` or `randi(imax,[m,n])` returns an m-by-n matrix.
`randi(imax,m,n,p,...)` or `randi(imax,[m,n,p,...])` returns an m-by-n-by-p-by-... array. `randi(imax,size(A))` returns an array the same size as A.

`r = randi([imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on imin:imax.

`r = randi(..., classname)` returns an array of integer values of class classname. `classname` does not support 64-bit integers.

---

**Note**
Note: The size inputs m, n, p, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randi` is determined by the internal state of the uniform pseudorandom number generator that underlies `rand`, `randi`, and `randn`. `randi` uses one uniform value from that default stream to generate each integer value. Control the default stream using its properties and methods. See `@RandStream` for details about the default stream.
Resetting the default stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties. Since the random number generator is initialized to the same state every time MATLAB software starts up, rand, randn, and randi will generate the same sequence of numbers in each session until the state is changed.

**Examples**

Generate integer values from the uniform distribution on the set 1:10.

\[
r = \text{randi}(10,100,1);
\]

Generate an integer array of integers drawn uniformly from 1:10.

\[
r = \text{randi}(10,100,1,'uint32');
\]

Generate integer values drawn uniformly from -10:10.

\[
r = \text{randi}([-10 10],100,1);
\]

Replace the default stream at MATLAB startup, using a stream whose seed is based on clock, so that randi will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session.

\[
\text{RandStream.setDefaultStream(RandStream('mt19937ar','seed',sum(100*clock)));
randi(100,1,5)}
\]

Save the current state of the default stream, generate 5 integer values, restore the state, and repeat the sequence.

\[
\text{defaultStream = RandStream.getDefaultStream;}
\text{savedState = defaultStream.State;}
\text{i1 = \text{randi}(10,1,5)}
\text{defaultStream.State = savedState;}
\text{i2 = \text{randi}(10,1,5) \%contains exactly the same values as i1}
\]
See Also

rand, randn, @RandStream, randi (RandStream), getDefaultStream (RandStream)
randi (RandStream)

**Purpose**
Uniformly distributed pseudorandom integers

**Class**
@RandStream

**Syntax**

```matlab
r = randi(s,imax,n)
randi(s,imax,m,n)
randi(s,imax,[m,n])
randi(s,imax,m,n,p,...)
randi(s,imax,[m,n,p,...])
randi(s,imax)
randi(s,imax,size(A))
r = randi(s,[imin,imax],...)
r = randi(..., classname)
```

**Description**

`r = randi(s,imax,n)` returns an n-by-n matrix containing pseudorandom integer values drawn from the discrete uniform distribution on 1:imax. `randi` draws those values from the random stream `s`. `randi(s,imax,m,n)` or `randi(s,imax,[m,n])` returns an m-by-n matrix. `randi(s,imax,m,n,p,...)` or `randi(s,imax,[m,n,p,...])` returns an m-by-n-by-p-by-... array. `randi(s,imax)` returns a scalar. `randi(s,imax,size(A))` returns an array the same size as `A`.

`r = randi(s,[imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on `imin:imax`.

`r = randi(..., classname)` returns an array of integer values of class `classname`. `classname` does not support 64-bit integers.

**Note**
The size inputs `m, n, p, ...` should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randi` is determined by the internal state of the random stream `s`. `randi` uses one uniform value from `s` to generate each integer value. Resetting `s` to the same fixed
state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

rand, @RandStream, rand (RandStream), randn (RandStream), randperm (RandStream)
**Purpose**

Normally distributed pseudorandom numbers

**Syntax**

\[ r = \text{randn}(n) \]
\[ \text{randn}(m,n) \]
\[ \text{randn}([m,n]) \]
\[ \text{randn}(m,n,p,...) \]
\[ \text{randn}([m,n,p,...]) \]
\[ \text{randn}(\text{size}(A)) \]
\[ r = \text{randn}(..., 'double') \]
\[ r = \text{randn}(..., 'single') \]

**Description**

\[ r = \text{randn}(n) \] returns an \( n \)-by-\( n \) matrix containing pseudorandom values drawn from the standard normal distribution. \[ \text{randn}(m,n) \] or \[ \text{randn}([m,n]) \] returns an \( m \)-by-\( n \) matrix. \[ \text{randn}(m,n,p,...) \] or \[ \text{randn}([m,n,p,...]) \] returns an \( m \)-by-\( n \)-by-\( p \)-by-... array. \text{randn} returns a scalar. \[ \text{randn}(\text{size}(A)) \] returns an array the same size as \( A \).

\[ r = \text{randn}(..., 'double') \] or \[ r = \text{randn}(..., 'single') \] returns an array of normal values of the specified class.

**Note**
The size inputs \( m, n, p, ... \) should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by \text{randn} is determined by the internal state of the uniform pseudorandom number generator that underlies \text{rand}, \text{randi}, and \text{randn}. \text{randn} uses one or more uniform values from that default stream to generate each normal value. Control the default stream using its properties and methods. See @RandStream for details about the default stream.

Resetting the default stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties. Since the random number generator is initialized to the same state every time MATLAB software starts up, \text{rand}, \text{randn}, and \text{randi} will...
generate the same sequence of numbers in each session until the state is changed.

**Note** In versions of MATLAB prior to 7.7, you controlled the internal state of the random number stream used by randn by calling randn directly with the 'seed' or 'state' keywords. That syntax is still supported for backwards compatibility, but is deprecated. For version 7.7, use the default stream as described in the @RandStream reference documentation.

**Examples**

Generate values from a normal distribution with mean 1 and standard deviation 2.

```matlab
r = 1 + 2.*randn(100,1);
```

Generate values from a bivariate normal distribution with specified mean vector and covariance matrix.

```matlab
mu = [1 2];
Sigma = [1 .5; .5 2]; R = chol(Sigma);
z = repmat(mu,100,1) + randn(100,2)*R;
```

Replace the default stream at MATLAB startup, using a stream whose seed is based on clock, so that randn will return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session.

```matlab
RandStream.setDefaultStream
    (RandStream('mt19937ar','seed',sum(100*clock)));
randn(1,5)
```

Save the current state of the default stream, generate 5 values, restore the state, and repeat the sequence.

```matlab
defaultStream = RandStream.getDefaultStream;
savedState = defaultStream.State;
```
z1 = randn(1,5)
defaultStream.State = savedState;
z2 = randn(1,5) % contains exactly the same values as z1

See Also
rand, randi, @RandStream, randn (RandStream), getDefaultStream (RandStream)
Purpose

Normally distributed pseudorandom numbers

Class

@RandStream

Syntax

randn(s,m,n)
randn(s,[m,n])
randn(s,m,n,p,...)
randn(s,[m,n,p,...])
randn(s)
randn(s,size(A))
r = randn(..., 'double')
r = randn(..., 'single')

Description

r = randn(s,n) returns an n-by-n matrix containing pseudorandom values drawn from the standard normal distribution. randn draws those values from the random stream s. randn(s,m,n) or randn(s,[m,n]) returns an m-by-n matrix. randn(s,m,n,p,...) or randn(s,[m,n,p,...]) returns an m-by-n-by-p-by-... array. randn(s) returns a scalar. randn(s,size(A)) returns an array the same size as A.

r = randn(..., 'double') or r = randn(..., 'single') returns an array of uniform values of the specified class.

Note

The size inputs m, n, p, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by randn is determined by the internal state of the random stream s. randn uses one or more uniform values from s to generate each normal value. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

randn, @RandStream, rand (RandStream), randi (RandStream)
**Purpose**  
Random permutation

**Syntax**  
\[ p = \text{randperm}(n) \]

**Description**  
\( p = \text{randperm}(n) \) returns a random permutation of the integers 1:n.

**Remarks**  
The `randperm` function calls `rand` and therefore changes the state of the default random number stream.

**Examples**  
`randperm(6)` might be the vector

\[ [3\ 2\ 6\ 4\ 1\ 5] \]

or it might be some other permutation of 1:6.

**See Also**  
`permute`
randperm (RandStream)

Class @RandStream

Syntax randperm(s,n)

Description randperm(s,n) generates a random permutation of the integers from 1 to n. For example, randperm(s,6) might be [2 4 5 6 1 3]. randperm(s,n) uses random values drawn from the random number stream s.

See Also permute, @RandStream
Purpose
Random number stream

Constructor
RandStream (RandStream)

Description
Pseudorandom numbers in MATLAB come from one or more random number streams. The simplest way to generate arrays of random numbers is to use rand, randn, or randi. These functions all rely on the same stream of uniform random numbers, known as the default stream. You can create other stream objects that act separately from the default stream, and you can use their rand, randi, or randn methods to generate arrays of random numbers. You can also create a random number stream and make it the default stream.

To create a single random number stream, use either the RandStream constructor or the RandStream.create factory method. To create multiple independent random number streams, use RandStream.create.

stream = RandStream.getDefaultStream returns the default random number stream, that is, the one currently used by the rand, randi, and randn functions.

prevstream = RandStream.setDefaultStream(stream) returns the current default stream, and designates the random number stream stream as the new default to be used by the rand, randi, and randn functions.

A random number stream s has properties that control its behavior. Access or assign to a property using p= s.Property or s.Property = p. The following table lists defined properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>(Read-only) Generator algorithm used by the stream. The list of possible generators is given by RandStream.list.</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Seed</td>
<td>(Read-only) Seed value used to create the stream.</td>
</tr>
<tr>
<td>NumStreams</td>
<td>(Read-only) Number of streams in the group in which the current stream was created.</td>
</tr>
<tr>
<td>StreamIndex</td>
<td>(Read-only) Index of the current stream from among the group of streams with which it was created.</td>
</tr>
<tr>
<td>State</td>
<td>Internal state of the generator. You should not depend on the format of this property. The value you assign to <code>S.State</code> must be a value read from <code>S.State</code> previously.</td>
</tr>
<tr>
<td>Substream</td>
<td>Index of the substream to which the stream is currently set. The default is 1. Multiple substreams are not supported by all generator types; the multiplicative lagged Fibonacci generator (<code>mlfg6331_64</code>) and combined multiple recursive generator (<code>mrg32k3a</code>) support multiple streams.</td>
</tr>
<tr>
<td>RandnAlg</td>
<td>Algorithm used by <code>randn(s, ...)</code> to generate normal pseudorandom values. Possible values are 'Ziggurat', 'Polar', or 'Inversion'.</td>
</tr>
</tbody>
</table>
### RandStream

<table>
<thead>
<tr>
<th><strong>Property</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Antithetic</td>
<td>Logical value indicating whether $S$ generates antithetic pseudorandom values. For uniform values, these are the usual values subtracted from 1. The default is false.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FullPrecision</td>
<td>Logical value indicating whether $S$ generates values using its full precision. Some generators can create pseudorandom values faster, but with fewer random bits, if FullPrecision is false. The default is true.</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th><strong>Method</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>RandStream</td>
<td>Create a random number stream</td>
</tr>
<tr>
<td>create</td>
<td>Create multiple independent random number streams</td>
</tr>
<tr>
<td>get</td>
<td>Get the properties of a random stream object</td>
</tr>
<tr>
<td>list</td>
<td>List available random number generator algorithms</td>
</tr>
<tr>
<td>getDefaultStream</td>
<td>Get the default random number stream</td>
</tr>
<tr>
<td>setDefaultStream</td>
<td>Set the default random number stream</td>
</tr>
<tr>
<td>reset</td>
<td>Reset a stream to its initial internal state</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>rand</td>
<td>Pseudorandom numbers from a uniform distribution</td>
</tr>
<tr>
<td>randn</td>
<td>Pseudorandom numbers from a standard normal distribution</td>
</tr>
<tr>
<td>randi</td>
<td>Pseudorandom integers from a uniform discrete distribution</td>
</tr>
<tr>
<td>randperm</td>
<td>Random permutation of a set of values</td>
</tr>
</tbody>
</table>

**See Also**
rand, randn, randi, rand (RandStream), randn (RandStream), randi (RandStream)
**Purpose**
Random number stream

**Class**
@RandStream

**Syntax**
s = RandStream('gentype')
[...]=RandStream('gentype','param1',val1,'param2',val2,...)

**Description**
s = RandStream('gentype') creates a random number stream that uses the uniform pseudorandom number generator algorithm specified by gentype.[...]=RandStream('gentype','param1',val1,'param2',val2,...) allows you to specify optional parameter name/value pairs to control creation of the stream. Options for gentype are given by RandStream.list.

Parameters are for RandStream are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed</td>
<td>Nonnegative scalar integer with which to initialize all streams. Default is 0. Seeds must be an integer between 0 and .</td>
</tr>
<tr>
<td>RandnAlg</td>
<td>Algorithm used by randn(s, ...) to generate normal pseudorandom values. Possible values are 'Ziggurat', 'Polar', or 'Inversion'.</td>
</tr>
</tbody>
</table>

**Examples**
Construct a random stream object using the combined multiple recursive generator and generate 5 uniformly distributed values from that stream.

```matlab
stream=RandStream('mrg32k3a');
rand(stream,1,5)
```
Construct a random stream object using the multiplicative lagged Fibonacci generator and generate 5 normally distributed values using the polar algorithm.

```matlab
stream=RandStream('mlfg6331_64','RandnAlg','Polar');
randn(stream,1,5)
```

**See Also**

@RandStream, rand (RandStream), randn (RandStream), randi (RandStream), getDefaultStream (RandStream)
**Purpose**

Rank of matrix

**Syntax**

```
k = rank(A)
k = rank(A,tol)
```

**Description**

The `rank` function provides an estimate of the number of linearly independent rows or columns of a full matrix.

- `k = rank(A)` returns the number of singular values of `A` that are larger than the default tolerance, `max(size(A))*eps(norm(A))`.
- `k = rank(A,tol)` returns the number of singular values of `A` that are larger than `tol`.

**Remark**

Use `sprank` to determine the structural rank of a sparse matrix.

**Algorithm**

There are a number of ways to compute the rank of a matrix. MATLAB software uses the method based on the singular value decomposition, or SVD. The SVD algorithm is the most time consuming, but also the most reliable.

The `rank` algorithm is

```
s = svd(A);
tol = max(size(A))*eps(max(s));
r = sum(s > tol);
```

**See Also**

`sprank`

**References**

**Purpose**

Rational fraction approximation

**Syntax**

\[
\begin{align*}
[N,D] &= \text{rat}(X) \\
[N,D] &= \text{rat}(X,\text{tol}) \\
\text{rat}(X) \\
S &= \text{rats}(X,\text{strlen}) \\
S &= \text{rats}(X)
\end{align*}
\]

**Description**

Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. The \text{rat} function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The \text{rats} function calls \text{rat}, and returns strings.

\[
\begin{align*}
[N,D] &= \text{rat}(X) \text{ returns arrays } N \text{ and } D \text{ so that } \frac{N}{D} \text{ approximates } X \text{ to within the default tolerance, } 1.e^{-6} \text{norm}(X(:,1)). \\
[N,D] &= \text{rat}(X,\text{tol}) \text{ returns } \frac{N}{D} \text{ approximating } X \text{ to within } \text{tol}. \\
\text{rat}(X) \text{, with no output arguments, simply displays the continued fraction.} \\
S &= \text{rats}(X,\text{strlen}) \text{ returns a string containing simple rational approximations to the elements of } X. \text{ Asterisks are used for elements that cannot be printed in the allotted space, but are not negligible compared to the other elements in } X. \text{ strlen is the length of each string element returned by the } \text{rats} \text{ function. The default is } \text{strlen} = 13, \text{ which allows 6 elements in 78 spaces.} \\
S &= \text{rats}(X) \text{ returns the same results as those printed by MATLAB with format } \text{rat}.
\end{align*}
\]

**Examples**

Ordinarily, the statement

\[
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7
\]

produces

\[
s =
\]
0.7595

However, with

format rat

or with

rats(s)

the printed result is

\[ s = \frac{319}{420} \]

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity \( s \) is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement \( \text{rat}(s) \) produces

\[ 1 + \frac{1}{(-4 + \frac{1}{(-6 + \frac{1}{(-3 + \frac{1}{(-5)})}})}) \]

And the statement

\[ [n,d] = \text{rat}(s) \]

produces

\[ n = 319, \quad d = 420 \]

The mathematical quantity \( \pi \) is certainly not a rational number, but the MATLAB quantity \( \pi \) that approximates it is a rational number. \( \pi \) is the ratio of a large integer and \( 2^{52} \):

\[ 14148475504056880/4503599627370496 \]
However, this is not a simple rational number. The value printed for pi with format rat, or with rats(pi), is

\[
\frac{355}{113}
\]

This approximation was known in Euclid’s time. Its decimal representation is

\[
3.14159292035398
\]

and so it agrees with pi to seven significant figures. The statement

\[
rat(pi)
\]

produces

\[
3 + \frac{1}{(7 + \frac{1}{(16)})}
\]

This shows how the 355/113 was obtained. The less accurate, but more familiar approximation 22/7 is obtained from the first two terms of this continued fraction.

**Algorithm**

The rat(X) function approximates each element of X by a continued fraction of the form

\[
\frac{n}{d} = d_1 + \frac{1}{d_2 + \frac{1}{d_3 + \frac{1}{\ldots + \frac{1}{d_k}}}}
\]

The \(d\)'s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when \(X = \sqrt{2}\). For \(x = \sqrt{2}\), the error with \(k\) terms is about \(2.68^*(.173)^k\), so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.
rat, rats

See Also  format
**Purpose**

Create rubberband box for area selection

**Syntax**

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

**Description**

`rbbox` initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure’s `CurrentPoint`, and begins tracking from this point.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current figure’s `Units` property, and measured from the lower left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure’s `Units` property, and measured from the lower left corner of the figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` figure units, `rbbox` redraws the rubberband box. The default `stepSize` is 1.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the `x` and `y` components of the lower left corner of the box, and `width` and `height` are the dimensions of the box.

**Remarks**

`rbbox` is useful for defining and resizing a rectangular region:
For box definition, initialRect is \([x \ y \ 0 \ 0]\), where \((x,y)\) is the figure’s CurrentPoint.

For box resizing, initialRect defines the rectangular region that you resize (e.g., a legend). fixedPoint is the corner diametrically opposite the tracking point.

rbbox returns immediately if a button is not currently pressed. Therefore, you use rbbox with waitforbuttonpress so that the mouse button is down when rbbox is called. rbbox returns when you release the mouse button.

**Examples**

Assuming the current view is `view(2)`, use the current axes’ CurrentPoint property to determine the extent of the rectangle in dataspace units:

```matlab
k = waitforbuttonpress;
point1 = get(gca,'CurrentPoint'); % button down detected
finalRect = rbbox; % return figure units
point2 = get(gca,'CurrentPoint'); % button up detected
point1 = point1(1,1:2); % extract x and y
point2 = point2(1,1:2);
p1 = min(point1,point2); % calculate locations
offset = abs(point1-point2); % and dimensions
x = [p1(1) p1(1)+offset(1) p1(1)+offset(1) p1(1) p1(1)];
y = [p1(2) p1(2) p1(2)+offset(2) p1(2)+offset(2) p1(2)];
hold on
axis manual
plot(x,y) % redraw in dataspace units
```

**See Also**

axis, dragrect, waitforbuttonpress

“View Control” on page 1-106 for related functions
Purpose
Matrix reciprocal condition number estimate

Syntax
\( c = \text{rcond}(A) \)

Description
\( c = \text{rcond}(A) \) returns an estimate for the reciprocal of the condition of \( A \) in 1-norm using the LAPACK condition estimator. If \( A \) is well conditioned, \( \text{rcond}(A) \) is near 1.0. If \( A \) is badly conditioned, \( \text{rcond}(A) \) is near 0.0. Compared to \( \text{cond} \), \( \text{rcond} \) is a more efficient, but less reliable, method of estimating the condition of a matrix.

Algorithm
For full matrices \( A \), \text{rcond} uses the LAPACK routines listed in the following table to compute the estimate of the reciprocal condition number.

<table>
<thead>
<tr>
<th></th>
<th>Real</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>A double</td>
<td>DLANGE, DGETRF, DGETRF, DGECON</td>
<td>ZLANGE, ZGETRF, ZGETRF, ZGECON</td>
</tr>
<tr>
<td>A single</td>
<td>SLANGE, SGETRF, SGECON</td>
<td>CLANGE, CGETRF, CGETRF, CGECON</td>
</tr>
</tbody>
</table>

See Also
\( \text{cond, condest, norm, normest, rank, svd} \)

References
Purpose
Read video frame data from multimedia reader object

Syntax
```
video = read(obj)
video = read(obj, index)
```

Description
```
video = read(obj) reads in video frames from the associated file. video is an H-by-W-by-B-by-F matrix where H is the image frame height, W is the image frame width, B is the number of bands in the image (e.g., 3 for RGB), and F is the number of frames read in. The default behavior is to read in all frames unless an index is specified. The type of data returned is always UINT8 data representing RGB24 video frames.
video = read(obj, index) performs the same operation, but reads only the frame(s) specified by index, where the first frame number is 1. index can be a single index, or a two-element array representing an index range of the video stream.
```

For example, read only the first frame:
```
    video = read(obj, 1);
```

Read the first 10 frames:
```
    video = read(obj, [1 10]);
```

You can use Inf to represent the last frame in the file:
```
    video = read(obj, Inf);
```

Read from frame 50 through the end of the file:
```
    video = read(obj, [50 Inf]);
```

If an invalid index is specified, the MATLAB software throws an error.

Examples
Construct a multimedia reader object associated with file xylophone.mpg and with the user tag property set to 'myreader1'.

2-3024
readerobj = mmreader('xylophone.mpg', 'tag', 'myreader1');

Read in all video frames from the file.
vidFrames = read(readerobj);

Determine the number of frames in the file.
numFrames = get(readerobj, 'NumberOfFrames');

Create a MATLAB movie struct from the video frames.
for k = 1 : numFrames
    mov(k).cdata = vidFrames(:,:,,:,k);
    mov(k).colormap = [];
end

Create a figure.
hf = figure;

Resize the figure based on the video’s width and height.
set(hf, 'position', [150 150 readerobj.Width readerobj.Height])

Play back the movie once at the video’s frame rate.
movie(hf, mov, 1, readerobj.FrameRate);

See Also
get, mmreader, movie, set
**Purpose**
Read data asynchronously from device

**Syntax**
readasync(obj)
readasync(obj,size)

**Description**
readasync(obj) initiates an asynchronous read operation on the serial port object, obj.

readasync(obj,size) asynchronously reads, at most, the number of bytes given by size. If size is greater than the difference between the InputBufferSize property value and the BytesAvailable property value, an error is returned.

**Remarks**
Before you can read data, you must connect obj to the device with the fopen function. A connected serial port object has a Status property value of open. An error is returned if you attempt to perform a read operation while obj is not connected to the device.

You should use readasync only when you configure the ReadAsyncMode property to manual. readasync is ignored if used when ReadAsyncMode is continuous.

The TransferStatus property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the stopasync function.

You can monitor the amount of data stored in the input buffer with the BytesAvailable property. Additionally, you can use the BytesAvailableFcn property to execute an M-file callback function when the terminator or the specified amount of data is read.

**Rules for Completing an Asynchronous Read Operation**
An asynchronous read operation with readasync completes when one of these conditions is met:

- The terminator specified by the Terminator property is read.
• The time specified by the Timeout property passes.
• The specified number of bytes is read.
• The input buffer is filled (if size is not specified).

Because readasync checks for the terminator, this function can be slow. To increase speed, you might want to configure ReadAsyncMode to continuous and continuously return data to the input buffer as soon as it is available from the device.

**Example**

This example creates the serial port object s on a Windows platform. It connects s to a Tektronix TDS 210 oscilloscope, configures s to read data asynchronously only if readasync is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```matlab
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using readasync. When the read operation is complete, return the data to the MATLAB workspace using fscanf.

```matlab
readasync(s)
s.BytesAvailable
ans =
15
out = fscanf(s)
out =
2.0399999619E0
fclose(s)
```
# readasync

## See Also

- Functions
  - fopen, stopasync

## Properties

- BytesAvailable, BytesAvailableFcn, ReadAsyncMode, Status, TransferStatus
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Real part of complex number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>$X = \text{real}(Z)$</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>$X = \text{real}(Z)$ returns the real part of the elements of the complex array $Z$.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>$\text{real}(2+3i)$ is 2.</td>
</tr>
</tbody>
</table>
Purpose
Natural logarithm for nonnegative real arrays

Syntax
Y = reallog(X)

Description
Y = reallog(X) returns the natural logarithm of each element in array X. Array X must contain only nonnegative real numbers. The size of Y is the same as the size of X.

Examples
M = magic(4)

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

reallog(M)

ans =
    2.7726   0.6931   1.0986   2.5649
    1.6094   2.3979   2.3026   2.0794
    2.1972   1.9459   1.7918   2.4849
    1.3863   2.6391   2.7081   0

See Also
log, realpow, realsqrt
### Purpose
Largest positive floating-point number

### Syntax
\n\text{n} = \text{realmax}

### Description
\n\text{n} = \text{realmax} \text{ returns the largest floating-point number representable on your computer. Anything larger overflows.}

\text{realmax('double')} \text{ is the same as realmax with no arguments.}

\text{realmax('single')} \text{ is the largest single precision floating point number representable on your computer. Anything larger overflows to single(Inf).}

### Examples
\text{realmax is one bit less than } 2^{1024} \text{ or about } 1.7977e+308.

### Algorithm
The \text{realmax} function is equivalent to \text{pow2}(2 - \text{eps}, \text{maxexp}), where \text{maxexp} is the largest possible floating-point exponent.

Execute \text{type realmax} to see \text{maxexp} for various computers.

### See Also
\text{eps, realmin, intmax}
**Purpose**
Smallest positive normalized floating-point number

**Syntax**

```
n = realmin
```

**Description**

```
n = realmin returns the smallest positive normalized floating-point number on your computer. Anything smaller underflows or is an IEEE “denormal.”
```

REALMIN('double') is the same as REALMIN with no arguments.

REALMIN('single') is the smallest positive normalized single precision floating point number on your computer.

**Examples**

```
realmin is 2^(-1022) or about 2.2251e-308.
```

**Algorithm**

The realmin function is equivalent to `pow2(1,minexp)` where `minexp` is the smallest possible floating-point exponent.

Execute `type realmin` to see `minexp` for various computers.

**See Also**

```latex
eps, realmax, intmin
```
Purpose
Array power for real-only output

Syntax
\[ Z = \text{realpow}(X,Y) \]

Description
\[ Z = \text{realpow}(X,Y) \] raises each element of array \( X \) to the power of its corresponding element in array \( Y \). Arrays \( X \) and \( Y \) must be the same size. The range of \text{realpow} is the set of all real numbers, i.e., all elements of the output array \( Z \) must be real.

Examples
\[ X = -2*\text{ones}(3,3) \]

\[
X = \\
\begin{array}{ccc}
-2 & -2 & -2 \\
-2 & -2 & -2 \\
-2 & -2 & -2 \\
\end{array}
\]

\[ Y = \text{pascal}(3) \]

\[
\text{ans} = \\
\begin{array}{ccc}
1 & 1 & 1 \\
1 & 2 & 3 \\
1 & 3 & 6 \\
\end{array}
\]

\[ \text{realpow}(X,Y) \]

\[
\text{ans} = \\
\begin{array}{ccc}
-2 & -2 & -2 \\
-2 & 4 & -8 \\
-2 & -8 & 64 \\
\end{array}
\]

See Also
reallog, realsqrt, \(^\text{array power operator}\)
Purpose
Square root for nonnegative real arrays

Syntax
Y = realsqrt(X)

Description
Y = realsqrt(X) returns the square root of each element of array X. Array X must contain only nonnegative real numbers. The size of Y is the same as the size of X.

Examples
M = magic(4)

M =

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

realsqrt(M)

ans =

4.0000  1.4142  1.7321  3.6056
2.2361  3.3166  3.1623  2.8284
3.0000  2.6458  2.4495  3.4641
2.0000  3.7417  3.8730  1.0000

See Also
reallog, realpow, sqrt, sqrtm
**Purpose**
Record data and event information to file

**Syntax**
```
record(obj)
record(obj,'switch')
```

**Description**
- `record(obj)` toggles the recording state for the serial port object, `obj`.
- `record(obj,'switch')` initiates or terminates recording for `obj`. `switch` can be on or off. If `switch` is on, recording is initiated. If `switch` is off, recording is terminated.

**Remarks**
Before you can record information to disk, `obj` must be connected to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to Debugging: Recording Information to Disk.

**Example**
This example creates the serial port object `s` on a Windows platform. It connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```matlab
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
s.RecordName = 'MySerialFile.txt';
record(s,'on')
fprintf(s,'*IDN?')
out = fscanf(s);
record(s,'off')
```
fclose(s)

See Also

Functions
fclose, fopen

Properties
RecordDetail, RecordMode, RecordName, RecordStatus, Status
Purpose  
Create 2-D rectangle object

Syntax  
rectangle
rectangle('Position',[x,y,w,h])
rectangle(...,'Curvature',[x,y])
h = rectangle(...)

Description  
rectangle draws a rectangle with Position [0,0,1,1] and Curvature [0,0] (i.e., no curvature).

rectangle('Position',[x,y,w,h]) draws the rectangle from the point x,y and having a width of w and a height of h. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the x and y axes. You can do this with the command axis equal or daspect([1,1,1]).

rectangle(...,'Curvature',[x,y]) specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature x is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature y is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of x and y can range from 0 (no curvature) to 1 (maximum curvature). A value of [0,0] creates a rectangle with square sides. A value of [1,1] creates an ellipse. If you specify only one value for Curvature, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

h = rectangle(...) returns the handle of the rectangle object created.

Remarks  
Rectangle objects are 2-D and can be drawn in an axes only if the view is [0 90] (i.e., view(2)). Rectangles are children of axes and are defined in coordinates of the axes data.
Examples

This example sets the data aspect ratio to \([1,1,1]\) so that the rectangle is displayed in the specified proportions (\texttt{daspect}). Note that the horizontal and vertical curvature can be different. Also, note the effects of using a single value for \texttt{Curvature}.

\begin{verbatim}
rectangle('Position',[0.59,0.35,3.75,1.37],...
    'Curvature',[0.8,0.4],...
    'LineWidth',2,'LineStyle','--')

daspect([1,1,1])
\end{verbatim}

Specifying a single value of \([0.4]\) for \texttt{Curvature} produces
A Curvature of [1] produces a rectangle with the shortest side completely round:

This example creates an ellipse and colors the face red.

```matlab
rectangle('Position',[1,2,5,10],'Curvature',[1,1],...
    'FaceColor','r')
daspect([1,1,1])
xlim([0,7])
```
You can set default rectangle properties on the axes, figure, and root object levels:

```matlab
ylim([1,13])
```

```matlab
set(0,'DefaultRectangleProperty',PropertyValue...)
set(gcf,'DefaultRectangleProperty',PropertyValue...)
set(gca,'DefaultRectangleProperty',PropertyValue...)
```

where `Property` is the name of the rectangle property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.
See Also

line, patch

“Object Creation” on page 1-101 for related functions

See the annotation function for information about the rectangle annotation object.

Rectangle Properties for property descriptions
Rectangle Properties

**Purpose**

Define rectangle properties

**Modifying Properties**

You can set and query graphics object properties in two ways:

- “The Property Editor” is an interactive tool that enables you to see and change object property values.
- The set and get commands enable you to set and query the values of properties.

To change the default values of properties, see “Setting Default Property Values”.

See “Core Graphics Objects” for general information about this type of object.

**Rectangle Property Descriptions**

This section lists property names along with the type of values each accepts. Curly braces {} enclose default values.

**Annotation**

hg.Annotation object Read Only

*Control the display of rectangle objects in legends.* The Annotation property enables you to specify whether this rectangle object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the rectangle object is displayed in a figure legend:
### IconDisplayStyle Property

<table>
<thead>
<tr>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Represent this rectangle object in a legend (default)</td>
</tr>
<tr>
<td>off</td>
<td>Do not include this rectangle object in a legend</td>
</tr>
<tr>
<td>children</td>
<td>Same as on because rectangle objects do not have children</td>
</tr>
</tbody>
</table>

#### Setting the IconDisplayStyle property

These commands set the IconDisplayStyle of a graphics object with handle `hobj` to `off`:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

#### Using the IconDisplayStyle property

See “Controlling Legends” for more information and examples.

### BeingDeleted

<table>
<thead>
<tr>
<th>on</th>
<th>{off} read only</th>
</tr>
</thead>
</table>

*This object is being deleted.* The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. The MATLAB software sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted,
and therefore, can check the object's BeingDeleted property before acting.

**BusyAction**

cancel | {queue}

*Callback routine interruption.* The **BusyAction** property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the **Interruptible** property of the object whose callback is executing is set to **on** (the default), then interruption occurs at the next point where the event queue is processed. If the **Interruptible** property is set to **off**, the **BusyAction** property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- **cancel** — Discard the event that attempted to execute a second callback routine.
- **queue** — Queue the event that attempted to execute a second callback routine until the current callback finishes.

**ButtonDownFcn**

function handle, cell array containing function handle and additional arguments, or string (not recommended)

*Button press callback function.* A callback function that executes whenever you press a mouse button while the pointer is over the rectangle object.

See the figure’s **SelectionType** property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property)
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
    sel_typ = get(gca,'SelectionType')
    switch sel_typ
    case 'normal'
        disp('User clicked left-mouse button')
        set(src,'Selected','on')
    case 'extend'
        disp('User did a shift-click')
        set(src,'Selected','on')
    case 'alt'
        disp('User did a control-click')
        set(src,'Selected','on')
        set(src,'SelectionHighlight','off')
    end
end

Suppose h is the handle of a rectangle object and that the button_down function is on your MATLAB path. The following statement assigns the function above to the ButtonDownFcn:

    set(h,'ButtonDownFcn',@button_down)

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

Children
vector of handles

The empty matrix; rectangle objects have no children.

Clipping
{on} | off

*Clipping mode.* MATLAB clips rectangles to the axes plot box by default. If you set Clipping to off, rectangles are displayed outside the axes plot box. This can occur if you create a rectangle,
set hold to on, freeze axis scaling (axis set to manual), and then create a larger rectangle.

**CreateFcn**
function handle, cell array containing function handle and additional arguments, or string (not recommended)

*Callback function executed during object creation.* This property defines a callback function that executes when MATLAB creates a rectangle object. You must define this property as a default value for rectangles or in a call to the `rectangle` function to create a new rectangle object. For example, the statement

```matlab
set(0,'DefaultRectangleCreateFcn',@rect_create)
```

defines a default value for the rectangle `CreateFcn` property on the root level that sets the axes `DataAspectRatio` whenever you create a rectangle object. The callback function must be on your MATLAB path when you execute the above statement.

```matlab
function rect_create(src,evnt)
   % src - the object that is the source of the event
   % evnt - empty for this property
   axh = get(src,'Parent');
   set(axh,'DataAspectRatio',[1,1,1]))
end
```

MATLAB executes this function after setting all rectangle properties. Setting this property on an existing rectangle object has no effect. The function must define at least two input arguments (handle of object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.
See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

Curvature
one- or two-element vector \([x, y]\)

*Amount of horizontal and vertical curvature.* This property specifies the curvature of the rectangle sides, which enables the shape of the rectangle to vary from rectangular to ellipsoidal. The horizontal curvature \(x\) is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature \(y\) is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of \(x\) and \(y\) can range from 0 (no curvature) to 1 (maximum curvature). A value of \([0, 0]\) creates a rectangle with square sides. A value of \([1, 1]\) creates an ellipse. If you specify only one value for Curvature, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

DeleteFcn
function handle, cell array containing function handle and additional arguments, or string (not recommended)

*Delete rectangle callback function.* A callback function that executes when you delete the rectangle object (e.g., when you issue a delete command or clear the axes cla or figure clf). For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
    obj_tp = get(src,'Type');
    disp([obj_tp, ' object deleted'])
    disp('Its user data is:')
    disp(get(src,'UserData'))
```
MATLAB executes the function before deleting the object’s properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose DeleteFcn is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root CallbackObject property, which you can query using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

DisplayName
string (default is empty string)

*String used by legend for this rectangle object.* The `legend` function uses the string defined by the `DisplayName` property to label this rectangle object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this rectangle object’s corresponding string and that string is used for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where `n` is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`. 
Rectangle Properties

- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See “Controlling Legends” for more examples.

**EdgeColor**

\{ColorSpec\} | none

*Color of the rectangle edges.* This property specifies the color of the rectangle edges as a color or specifies that no edges be drawn.

**EraseMode**

\{normal\} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase rectangle objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- **none** — Do not erase the rectangle when it is moved or destroyed. While the object is still visible on the screen after erasing with **EraseMode none**, you cannot print it because MATLAB stores no information about its former location.

- **xor** — Draw and erase the rectangle by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the rectangle. However, the rectangle’s color depends on the color of whatever is beneath it on the display.

- **background** — Erase the rectangle by drawing it in the axes background Color, or the figure background Color if the axes
Rectangle Properties

Color is set to none. This damages objects that are behind the erased rectangle, but rectangles are always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the EraseMode of all objects is normal. This means graphics objects created with EraseMode set to none, xor, or background can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB getframe command or other screen capture application to create an image of a figure containing nonnormal mode objects.

FaceColor

ColorSpec | {none}

Color of rectangle face. This property specifies the color of the rectangle face, which is not colored by default.

HandleVisibility

{on} | callback | off

Control access to object’s handle by command-line users and GUIs. This property determines when an object’s handle is visible in its parent’s list of children. HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from
the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle’s visibility is restricted using `callback` or `off`, the object’s handle does not appear in its parent’s `Children` property, figures do not appear in the root’s `CurrentFigure` property, objects do not appear in the root’s `CallbackObject` property or in the figure’s `CurrentObject` property, and axes do not appear in their parent’s `CurrentAxes` property.

You can set the `Root` `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties and pass it to any function that operates on handles.

\[
\text{HitTest} \
\{\text{on}\} \mid \text{off}
\]

*Selectable by mouse click.* `HitTest` determines if the rectangle can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on
the rectangle. If HitTest is off, clicking the rectangle selects the object below it (which may be the axes containing it).

**Interruptible**

{on} | off

*Callback routine interruption mode.* The Interruptible property controls whether a rectangle callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the ButtonDownFcn are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback routine only when it encounters a drawnow, figure, getframe, or pause command in the routine.

**LineStyle**

{[{-} | – | : | -. | none}

*Line style of rectangle edge.* This property specifies the line style of the edges. The available line styles are

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>--</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

**LineWidth**

scalar

*The width of the rectangle edge line.* Specify this value in points (1 point = \(1/72 \text{ inch}\)). The default LineWidth is 0.5 points.

**Parent**

handle of axes, hggroup, or hgtransform
Parent of rectangle object. This property contains the handle of the rectangle object’s parent. The parent of a rectangle object is the axes, hggroup, or hgtransform object that contains it.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

Position
four-element vector \([x, y, \text{width}, \text{height}]\)

Location and size of rectangle. This property specifies the location and size of the rectangle in the data units of the axes. The point defined by \(x, y\) specifies one corner of the rectangle, and \(\text{width}\) and \(\text{height}\) define the size in units along the \(x\)-and \(y\)-axes respectively.

Selected
on \| off

Is object selected? When this property is on MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

SelectionHighlight
{on} \| off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing handles at each vertex. When SelectionHighlight is off, MATLAB does not draw the handles.

Tag
string

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as
global variables or pass them as arguments between callback routines. You can define Tag as any string.

**Type**

string (read only)

*Class of graphics object.* For rectangle objects, Type is always the string 'rectangle'.

**UIContextMenu**

handle of a uicontextmenu object

*Associate a context menu with the rectangle.* Assign this property the handle of a uicontextmenu object created in the same figure as the rectangle. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click over the rectangle.

**UserData**

matrix

*User-specified data.* Any data you want to associate with the rectangle object. MATLAB does not use this data, but you can access it using the set and get commands.

**Visible**

{on}  |  off

*Rectangle visibility.* By default, all rectangles are visible. When set to off, the rectangle is not visible, but still exists, and you can get and set its properties.
Purpose

Rectangle intersection area

Syntax

area = rectint(A,B)

Description

area = rectint(A,B) returns the area of intersection of the rectangles specified by position vectors A and B.

If A and B each specify one rectangle, the output area is a scalar.

A and B can also be matrices, where each row is a position vector. area is then a matrix giving the intersection of all rectangles specified by A with all the rectangles specified by B. That is, if A is n-by-4 and B is m-by-4, then area is an n-by-m matrix where area(i,j) is the intersection area of the rectangles specified by the ith row of A and the jth row of B.

Note A position vector is a four-element vector [x,y,width,height], where the point defined by x and y specifies one corner of the rectangle, and width and height define the size in units along the x and y axes respectively.

See Also

polyarea
**Purpose**

Set option to move deleted files to recycle folder

**Syntax**

S = recycle
S = recycle state
S = recycle('state')

**Description**

S = recycle returns a character array S that shows the current state of the file recycling option in the MATLAB application. This state can be either on or off. When file recycling is on, MATLAB moves all files that you delete with the delete function to either the recycle bin on Microsoft Windows or Apple Macintosh platforms, or to a temporary directory on UNIX platforms. (To locate this directory on UNIX platforms, see the Remarks section below.) When file recycling is off, any files you delete are actually removed from the system. The default recycle state is off. You can turn on recycling for all of your MATLAB sessions using preferences. Select File > Preferences > General, and for Default behavior of the delete function, select Move files to the Recycle Bin. For more information, click the Help button in General Preferences.

S = recycle state sets the recycle option for MATLAB to the specified state, either on or off. Return value S shows the previous recycle state.

S = recycle('state') is the function format.

**Remarks**

On UNIX platforms, you can locate the system temporary directory by entering the MATLAB function tempdir. The recycle directory is a subdirectory of this temporary directory, and is named according to the format

```
MATLAB_Files_<day>-<mo>-<yr>_<hr>_<min>_<sec>
```

For example, files recycled on a UNIX platform at 2:09:28 in the afternoon of November 9, 2007 are copied to a directory named

24. UNIX is a registered trademark of The Open Group in the United States and other countries.
You can recycle files that are stored on your local computer system, but not files that you access over a network connection. On Windows platforms, when you use the `delete` function on files accessed over a network, MATLAB removes the file entirely, regardless of the recycle state.

**Examples**

Start from a state where file recycling has been turned off. Check the current recycle state:

```plaintext
close all
recycle
ans =
    off
```

Turn file recycling on. Delete a file and verify that it is transferred to the recycle bin or temporary folder:

```plaintext
recycle on;
delete myfile.txt
```

**See Also**

- `delete`, `dir`, `ls`, `fileparts`, `mkdir`, `rmdir`
- “Managing Files and Working with the Current Directory”
Purpose
Reduce number of patch faces

Syntax
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
nfv = reducepatch(p) or nfv = reducepatch(fv)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)

Description
reducepatch(p,r) reduces the number of faces of the patch identified by handle p, while attempting to preserve the overall shape of the original object. The MATLAB software interprets the reduction factor r in one of two ways depending on its value:

- If r is less than 1, r is interpreted as a fraction of the original number of faces. For example, if you specify r as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If r is greater than or equal to 1, then r is the target number of faces. For example, if you specify r as 400, then the number of faces is reduced until there are 400 faces remaining.

nfv = reducepatch(p,r) returns the reduced set of faces and vertices but does not set the Faces and Vertices properties of patch p. The struct nfv contains the faces and vertices after reduction.

nfv = reducepatch(fv,r) performs the reduction on the faces and vertices in the struct fv.

nfv = reducepatch(p) or nfv = reducepatch(fv) uses a reduction value of 0.5.
reducepatch(...,'fast') assumes the vertices are unique and does not compute shared vertices.
reducepatch(...,'verbose') prints progress messages to the command window as the computation progresses.
\( nfv = \text{reducepatch}(f,v,r) \) performs the reduction on the faces in \( f \) and the vertices in \( v \).

\([nf,nv] = \text{reducepatch}(\ldots)\) returns the faces and vertices in the arrays \( nf \) and \( nv \).

**Remarks**

If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

The number of output triangles may not be exactly the number specified with the reduction factor argument \( (r) \), particularly if the faces of the original patch are not triangles.

**Examples**

This example illustrates the effect of reducing the number of faces to only 15% of the original value.

\[
[x,y,z,v] = \text{flow}; \\
p = \text{patch(isosurface}(x,y,z,v,-3)); \\
set(p,'facecolor','w','EdgeColor','b'); \\
daspect([1,1,1]) \\
view(3) \\
figure; \\
h = \text{axes}; \\
p2 = \text{copyobj}(p,h); \\
reducepatch(p2,0.15) \\
daspect([1,1,1]) \\
view(3)
\]
See Also

isosurface, isocaps, isonormals, smooth3, subvolume, reducevolume

“Volume Visualization” on page 1-108 for related functions

Vector Field Displayed with Cone Plots for another example
Purpose
Reduce number of elements in volume data set

Syntax
\[ [nx,ny,nz,nv] = \text{reducevolume}(X,Y,Z,V,[Rx,Ry,Rz]) \]
\[ [nx,ny,nz,nv] = \text{reducevolume}(V,[Rx,Ry,Rz]) \]
\[ nv = \text{reducevolume}(...) \]

Description
\[ [nx,ny,nz,nv] = \text{reducevolume}(X,Y,Z,V,[Rx,Ry,Rz]) \] reduces the number of elements in the volume by retaining every \( Rx \)th element in the \( x \) direction, every \( Ry \)th element in the \( y \) direction, and every \( Rz \)th element in the \( z \) direction. If a scalar \( R \) is used to indicate the amount or reduction instead of a three-element vector, the MATLAB software assumes the reduction to be \([R R R]\).

The arrays \( X, Y, \) and \( Z \) define the coordinates for the volume \( V \). The reduced volume is returned in \( nv \), and the coordinates of the reduced volume are returned in \( nx, ny, \) and \( nz \).

\[ [nx,ny,nz,nv] = \text{reducevolume}(V,[Rx,Ry,Rz]) \] assumes the arrays \( X, Y, \) and \( Z \) are defined as \( [X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p) \), where \( [m,n,p] = \text{size}(V) \).
\[ nv = \text{reducevolume}(...) \] returns only the reduced volume.

Examples
This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (squeeze) into three dimensions and then reduced (reducevolume) so that what remains is every fourth element in the \( x \) and \( y \) directions and every element in the \( z \) direction.
- The reduced data is smoothed (smooth3).
- The outline of the skull is an isosurface generated as a patch (p1) whose vertex normals are recalculated to improve the appearance when lighting is applied (patch, isosurface, isonormals).
- A second patch (p2) with an interpolated face color draws the end caps (FaceColor, isocaps).
- The view of the object is set (view, axis, daspect).
A 100-element grayscale colormap provides coloring for the end caps (colormap).

Adding a light to the right of the camera illuminates the object (camlight, lighting).

load mri
D = squeeze(D);
[x,y,z,D] = reducevolume(D,[4,4,1]);
D = smooth3(D);
p1 = patch(isosurface(x,y,z,D, 5,'verbose'),
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
    'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight; lighting gouraud

See Also
isosurface, isocaps, isonormals, smooth3, subvolume, reducepatch
“Volume Visualization” on page 1-108 for related functions
Purpose
Redraw current figure

Syntax
refresh
refresh(h)

Description
refresh erases and redraws the current figure.
refresh(h) redraws the figure identified by h.

See Also
“Figure Windows” on page 1-102 for related functions
**Purpose**
Refresh data in graph when data source is specified

**Syntax**
```
refreshdata
refreshdata(figure_handle)
refreshdata(object_handles)
refreshdata(object_handles,'workspace')
```

**Description**
`refreshdata` evaluates any data source properties (XDataSource, YDataSource, or ZDataSource) on all objects in graphs in the current figure. If the specified data source has changed, the MATLAB software updates the graph to reflect this change.

Note that the variable assigned to the data source property must be in the base workspace.

- `refreshdata(figure_handle)` refreshes the data of the objects in the specified figure.
- `refreshdata(object_handles)` refreshes the data of the objects specified in `object_handles` or the children of those objects. Therefore, `object_handles` can contain figure, axes, or plot object handles.
- `refreshdata(object_handles,'workspace')` enables you to specify whether the data source properties are evaluated in the base workspace or the workspace of the function in which `refreshdata` was called. The argument `workspace` is a string that can be
  - `base` — Evaluate the data source properties in the base workspace.
  - `caller` — Evaluate the data source properties in the workspace of the function that called `refreshdata`.

**Remarks**
The Linked Plots feature (see documentation for `linked`) sets up data sources for graphs and synchronizes them with the workspace variables they display. When you use this feature, you do not also need to call `refreshdata`, as it is essentially automatically triggered every time a data source changes.
If you are not using the Linked Plots feature, you need to set the XDataSource, YDataSource, and/or ZDataSource properties of a graph in order to use refreshdata. You can do that programmatically, as shown in the examples below, or use the Property Editor, one of the plotting tools. In the Property Editor, select the graph (e.g., a lineseries object) and type in (or select from the drop-down choices) the name(s) of the workspace variable(s) from which you want the plot to refresh, in the fields labelled X Data Source, Y Data Source, and/or Z Data Source. The call to refreshdata causes the graph to update.

**Examples**

Plot a sine wave, identify data sources, and then modify its YDataSource:

```matlab
x = 0:.1:8;
y = sin(x);
h = plot(x,y)
set(h,'YDataSource','y')
set(h,'XDataSource','x')
y = sin(x.^3);
refreshdata
```

Create a surface plot, identify a ZDataSource for it, and change the data to a different size.

```matlab
Z = peaks(5);
h = surf(Z)
set(h,'ZDataSource','Z')
pause(3)
Z = peaks(25);
refreshdata
```

**See Also**
The [X,Y,Z]DataSource properties of plot objects.
Purpose

Match regular expression

Syntax

```matlab
regexp('str', 'expr')
[start_idx, end_idx, extents, matches, tokens, names, splits] = regexp('str', 'expr')
[v1, v2, ...] = regexp('str', 'expr', q1, q2, ...)
[v1 v2 ...] = regexp('str', 'expr', ..., options)
```

Each of these syntaxes applies to both `regexp` and `regexpi`. The `regexp` function is case sensitive in matching regular expressions to a string, and `regexpi` is case insensitive.

Description

The following descriptions apply to both `regexp` and `regexpi`:

`regexp('str', 'expr')` returns a row vector containing the starting index of each substring of `str` that matches the regular expression string `expr`. If no matches are found, `regexp` returns an empty array. The `str` and `expr` arguments can also be cell arrays of strings. See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for more information.

To specify more than one string to parse or more than one expression to match, see the guidelines listed below under “Multiple Strings or Expressions” on page 2-3073.

`[start_idx, end_idx, extents, matches, tokens, names, splits] = regexp('str', 'expr')` returns up to six values, one for each output variable you specify, and in the default order (as shown in the table below).

**Note** The `str` and `expr` inputs are required and must be entered as the first and second arguments, respectively. Any other input arguments (all are described below) are optional and can be entered following the two required inputs in any order.
\[ v_1, v_2, \ldots \] = \text{regexp('str', 'expr', q1, q2, \ldots) \text{ returns up to six values, one for each output variable you specify, and ordered according to the order of the qualifier arguments, q1, q2, etc.}}

Return Values for Regular Expressions

<table>
<thead>
<tr>
<th>Default Order</th>
<th>Description</th>
<th>Qualifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Row vector containing the starting index of each substring of str that matches expr.</td>
<td>start</td>
</tr>
<tr>
<td>2</td>
<td>Row vector containing the ending index of each substring of str that matches expr.</td>
<td>end</td>
</tr>
<tr>
<td>3</td>
<td>Cell array containing the starting and ending indices of each substring of str that matches a token in expr. (This is a double array when used with 'once'.)</td>
<td>tokenExtents</td>
</tr>
<tr>
<td>4</td>
<td>Cell array containing the text of each substring of str that matches expr. (This is a string when used with 'once'.)</td>
<td>match</td>
</tr>
<tr>
<td>5</td>
<td>Cell array of cell arrays of strings containing the text of each token captured by regexp. (This is a cell array of strings when used with 'once'.)</td>
<td>tokens</td>
</tr>
<tr>
<td>6</td>
<td>Structure array containing the name and text of each named token captured by regexp. If there are no named tokens in expr, regexp returns a structure array with no fields. Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression (?&lt;tokenname&gt;).</td>
<td>names</td>
</tr>
<tr>
<td>7</td>
<td>Cell array containing those parts of the input string that are delimited by substrings returned when using the regexp 'match' option.</td>
<td>split</td>
</tr>
</tbody>
</table>
Tip When using the split option, `regexp` always returns one more string than it does with the match option. Also, you can always put the original input string back together from the substrings obtained from both split and match. See “Example 4 — Splitting the Input String” on page 2-3074.

```matlab
[v1 v2 ...] = regexp('str', 'expr', ..., options)
```
calls `regexp` with one or more of the nondefault options listed in the following table. These options must follow `str` and `expr` in the input argument list.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>See the section on “Modes” on page 2-3070 below.</td>
</tr>
<tr>
<td>'once'</td>
<td>Return only the first match found.</td>
</tr>
<tr>
<td>'warnings'</td>
<td>Display any hidden warning messages issued by MATLAB during the execution of the command. This option only enables warnings for the one command being executed. See “Example 11 — Displaying Parsing Warnings” on page 2-3080.</td>
</tr>
</tbody>
</table>

**Modes**

You can specify one or more of the following modes with the `regexp`, `regexpi`, and `regexprep` functions. You can enable or disable any of these modes using the mode specifier keyword (e.g., `'lineanchors'`) or the mode flag (e.g., `(?m)`). Both are shown in the tables that follow. Use the keyword to enable or disable the mode for the entire string being parsed. Use the flag to both enable and disable the mode for selected pieces of the string.

**Case-Sensitivity Mode**

Use the Case-Sensitivity mode to control whether or not MATLAB considers letter case when matching an expression to a string. “Example 7 — Using the Case-Sensitive Mode” on page 2-3077 illustrates this mode.
### Mode Keyword

<table>
<thead>
<tr>
<th>Mode Keyword</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'matchcase'</td>
<td>(?-i)</td>
<td>Letter case must match when matching patterns to a string. (The default for regexp).</td>
</tr>
<tr>
<td>'ignorecase'</td>
<td>(?i)</td>
<td>Do not consider letter case when matching patterns to a string. (The default for regexp).</td>
</tr>
</tbody>
</table>

### Dot Matching Mode

Use the Dot Matching mode to control whether or not MATLAB includes the newline (\n) character when matching the dot (.) metacharacter in a regular expression. “Example 8 — Using the Dot Matching Mode” on page 2-3078 illustrates the Dot Matching mode.

<table>
<thead>
<tr>
<th>Mode Keyword</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dotall'</td>
<td>(?s)</td>
<td>Match dot (.) in the pattern string with any character. (This is the default).</td>
</tr>
<tr>
<td>'dotexceptnewline'</td>
<td>(?-s)</td>
<td>Match dot in the pattern with any character that is not a newline.</td>
</tr>
</tbody>
</table>

### Anchor Type Mode

Use the Anchor Type mode to control whether MATLAB considers the ^ and $ metacharacters to represent the beginning and end of a string or the beginning and end of a line. “Example 9 — Using the Anchor Type Mode” on page 2-3078 illustrates the Anchor mode.
### Mode
**Keyword** | **Flag** | **Description**
---|---|---
'stringanchors' | (?-m) | Match the `^` and `$` metacharacters at the beginning and end of a string. (This is the default). 
'lineanchors' | (?m) | Match the `^` and `$` metacharacters at the beginning and end of a line.

### Spacing Mode
Use the Spacing mode to control how MATLAB interprets space characters and comments within the parsing string. Note that spacing mode applies to the parsing string (the second input argument that contains the metacharacters (e.g., `\w`) and not the string being parsed. “Example 10 — Using the Spacing Mode” on page 2-3079 illustrates the Spacing mode.

<table>
<thead>
<tr>
<th>Mode Keyword</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'literalspacing'</td>
<td>(?-x)</td>
<td>Parse space characters and comments (the <code>#</code> character and any text to the right of it) in the same way as any other characters in the string. (This is the default).</td>
</tr>
<tr>
<td>'freespacing'</td>
<td>(?x)</td>
<td>Ignore spaces and comments when parsing the string. (You must use <code>'\ '</code> and <code>'\#'</code> to match space and <code>#</code> characters.)</td>
</tr>
</tbody>
</table>

### Remarks
See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for a listing of all regular expression elements supported by MATLAB.
Multiple Strings or Expressions

Either the str or expr argument, or both, can be a cell array of strings, according to the following guidelines:

- If str is a cell array of strings, then each of the regexp outputs is a cell array having the same dimensions as str.
- If str is a single string but expr is a cell array of strings, then each of the regexp outputs is a cell array having the same dimensions as expr.
- If both str and expr are cell arrays of strings, these two cell arrays must contain the same number of elements.

Examples

Example 1 — Matching a Simple Pattern

Return a row vector of indices that match words that start with c, end with t, and contain one or more vowels between them. Make the matches insensitive to letter case (by using regexpi):

```matlab
str = 'bat cat can car COAT court cut ct CAT-scan';
regexpi(str, 'c[aeiou]+t')
ans =
   5    17    28    35
```

Example 2 — Parsing Multiple Input Strings

Return a cell array of row vectors of indices that match capital letters and white spaces in the cell array of strings str:

```matlab
str = {'Madrid, Spain' 'Romeo and Juliet' 'MATLAB is great'};
s1 = regexp(str, '[A-Z]');
s2 = regexp(str, '\s');

Capital letters, '[A-Z]', were found at these str indices:

```
Space characters, '\s', were found at these str indices:

```
s2{:}
ans =
  8
ans =
  6 10
ans =
  7 10
```

**Example 3 — Selecting Return Values**

Return the text and the starting and ending indices of words containing the letter x:

```matlab
str = 'regexp helps you relax';
[m s e] = regexp(str, '\w*x\w*', 'match', 'start', 'end')
m =
  'regexp' 'relax'
s =
  1 18
e =
  6 22
```

**Example 4 — Splitting the Input String**

Find the substrings delimited by the ^ character:

```matlab
s1 = ['Use REGEXP to split ^this string into ' ...
  'several ^individual pieces'];

s2 = regexp(s1, '\^', 'split');

s2(:)
an =
  'Use REGEXP to split '
```
'this string into several'
'individual pieces'

The split option returns those parts of the input string that are not returned when using the 'match' option. Note that when you match the beginning or ending characters in a string (as is done in this example), the first (or last) return value is always an empty string:

```matlab
str = 'She sells sea shells by the seashore.';

[matchstr splitstr] = regexp(str, '\[Ss]h.', 'match', ...
    'split')
```

```matlab
matchstr =
    'She'    'she'    'sho'
splitstr =
    ''  ' sells sea '  'lls by the sea'  're.'
```

For any string that has been split, you can reassemble the pieces into the initial string using the command

```matlab
j = [splitstr; [matchstr {''}]]; [j{;}]
```

```matlab
ans =
    She sells sea shells by the seashore.
```

**Example 5 — Using Tokens**

Search a string for opening and closing HTML tags. Use the expression `<(\w+)` to find the opening tag (e.g., `<tagname>`) and to create a token for it. Use the expression `</\1>` to find another occurrence of the same token, but formatted as a closing tag (e.g., '</tagname>):

```matlab
str = ['if <code>A</code> == x<sup>2</sup>, ' ...
    '<em>disp(x)</em>']
```

```matlab
str =
if <code>A</code> == x<sup>2</sup>, <em>disp(x)</em>
```

```matlab
expr = '<(\w+).*?>.*?</\1>,'
```
regexp, regexpi

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

tok{:}
ans =
    'code'
ans =
    'sup'
ans =
    'em'

mat{:}
ans =
    <code>A</code>
ans =
    <sup>2</sup>
ans =
    <em>disp(x)</em>

See “Tokens” in the MATLAB Programming Fundamentals documentation for information on using tokens.

**Example 6 — Using Named Capture**

Enter a string containing two names, the first and last names being in a different order:

```matlab
str = sprintf('John Davis
Rogers, James')
str =
    John Davis
    Rogers, James
```

Create an expression that generates first and last name tokens, assigning the names first and last to the tokens. Call `regexp` to get the text and names of each token found:

```matlab
expr = ...
    '(?<first>\w+)\s+(?<last>\w+)|(?<last>\w+)\s+(?<first>\w+)';

[tokens names] = regexp(str, expr, 'tokens', 'names');
Examine the tokens cell array that was returned. The first and last name tokens appear in the order in which they were generated: first name–last name, then last name–first name:

```matlab
tokens{:}
ans =
    'John'    'Davis'
ans =
    'Rogers'   'James'
```

Now examine the names structure that was returned. First and last names appear in a more usable order:

```matlab
names(:,1)
ans =
    first: 'John'
    last: 'Davis'

names(:,2)
ans =
    first: 'James'
    last: 'Rogers'
```

**Example 7 — Using the Case-Sensitive Mode**

Given a string that has both uppercase and lowercase letters,

```matlab
str = 'A string with UPPERCASE and lowercase text.';
```

Use the `regexp` default mode (case-sensitive) to locate only the lowercase instance of the word case:

```matlab
regexp(str, 'case', 'match')
an =
    'case'
```

Now disable case-sensitive matching to find both instances of case:

```matlab
regexp(str, 'case', 'ignorecase', 'match')
an =
```

```matlab
2-3077
```
'CASE'   'case'

Match 5 letters that are followed by 'CASE'. Use the (?-i) flag to turn on case-sensitivity for the first match and (?i) to turn it off for the second:

\[
M = \text{regexp}(\text{str}, \{\text{'}(?-i)\w{5}(?=CASE)'\, , \ldots \\
\text{'}(?i)\w{5}(?=CASE)'\}, \text{'match'});
\]

\[
M{:} \\
\text{ans} = \\
\text{'UPPER'} \\
\text{ans} = \\
\text{'UPPER' 'lower'}
\]

**Example 8 — Using the Dot Matching Mode**

Parse the following string that contains a newline (\n) character:

\[
\text{str} = \text{sprintf('abc\ndef')} \\
\text{str} = \\
\text{abc} \\
\text{def}
\]

When you use the default mode, dotall, MATLAB includes the newline in the characters matched:

\[
\text{regexp}(\text{str}, \text{'.'}, \text{'match'}) \\
\text{ans} = \\
\text{'a' 'b' 'c' [1x1 char] 'd' 'e' 'f'}
\]

When you use the dotexceptnewline mode, MATLAB skips the newline character:

\[
\text{regexp}(\text{str}, \text{'.'}, \text{'match'}, \text{'dotexceptnewline'}) \\
\text{ans} = \\
\text{'a' 'b' 'c' 'd' 'e' 'f'}
\]

**Example 9 — Using the Anchor Type Mode**

Given the following two-line string,
str = sprintf('%s
%s', 'Here is the first line', ...
    'followed by the second line')

str =
    Here is the first line
    followed by the second line

In string anchors mode, MATLAB interprets the $ metacharacter as an end-of-string specifier, and thus finds the last two words of the entire string:

    regexp(str, '\w+\W\w+$', 'match', 'stringanchors')
    ans =
          'second line'

While in line anchors mode, MATLAB interprets $ as an end-of-line specifier, and finds the last two words of each line:

    regexp(str, '\w+\W\w+$', 'match', 'lineanchors')
    ans =
          'first line'    'second line'

**Example 10 — Using the Spacing Mode**

Create a file called regexp_str.txt containing the following text.

```matlab
(?x)   # turn on freespacing.

# This pattern matches a string with a repeated letter.
\w*   # First, match any number of preceding word characters.

(    # Mark a token.
    # Match a character of any type.
)    # Finish capturing said token.
\1   # Backreference to match what token #1 matched.
\w*  # Finally, match the remainder of the word.
```
Because the first line enables freespacing mode, MATLAB ignores all spaces and comments that appear in the file. Here is the string to parse:

\[
\text{str = ['Looking for words with letters that ' ... 'appear twice in succession.']}\]

Use the pattern expression read from the file to find those words that have consecutive matching letters:

\[
\text{patt = fileread('regexp_str.txt');}
\text{regexp(str, patt, 'match')}
\]
\[
\text{ans =}
\text{'Looking' 'letters' 'appear' 'succession'}
\]

**Example 11 — Displaying Parsing Warnings**

To help debug problems in parsing a string with \texttt{regexp}, \texttt{regexpi}, or \texttt{regextprep}, use the \texttt{warnings} option to view all warning messages:

\[
\text{regexp('$.','[a-]', 'warnings')}
\]
\[
\text{Warning: Unbound range.}
\text{[a-]}
\]

**See Also**

\texttt{regextprep, regextpretranslate, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi}
**Purpose**
Replace string using regular expression

**Syntax**

\[
\begin{align*}
  s &= \text{regexprep('str', 'expr', 'repstr')} \\
  s &= \text{regexprep('str', 'expr', 'repstr', options)}
\end{align*}
\]

**Description**

\( s = \text{regexprep('str', 'expr', 'repstr')} \) replaces all occurrences of the regular expression \( expr \) in string \( str \) with the string \( repstr \). The new string is returned in \( s \). If no matches are found, return string \( s \) is the same as input string \( str \). You can use character representations (e.g., '\t' for tab, or '\n' for newline) in replacement string \( repstr \). See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for more information.

If \( str \) is a cell array of strings, then the \text{regexprep} return value \( s \) is always a cell array of strings having the same dimensions as \( str \).

To specify more than one expression to match or more than one replacement string, see the guidelines listed below under “Multiple Expressions or Replacement Strings” on page 2-3082.

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

\( s = \text{regexprep('str', 'expr', 'repstr', options}) \) By default, \text{regexprep} replaces all matches and is case sensitive. You can use one or more of the following options with \text{regexprep}.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>See mode descriptions on the \text{regexp} reference page.</td>
</tr>
<tr>
<td>( N )</td>
<td>Replace only the ( N )th occurrence of ( expr ) in ( str ).</td>
</tr>
<tr>
<td>'once'</td>
<td>Replace only the first occurrence of ( expr ) in ( str ).</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>'ignorecase'</td>
<td>Ignore case when matching and when replacing.</td>
</tr>
<tr>
<td>'preservecase'</td>
<td>Ignore case when matching (as with 'ignorecase'), but override the case of replace characters with the case of corresponding characters in str when replacing.</td>
</tr>
<tr>
<td>'warnings'</td>
<td>Display any hidden warning messages issued by MATLAB during the execution of the command. This option only enables warnings for the one command being executed.</td>
</tr>
</tbody>
</table>

**Remarks**

See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for a listing of all regular expression metacharacters supported by MATLAB.

**Multiple Expressions or Replacement Strings**

In the case of multiple expressions and/or replacement strings, `regexprep` attempts to make all matches and replacements. The first match is against the initial input string. Successive matches are against the string resulting from the previous replacement.

The `expr` and `replstr` inputs follow these rules:

- If `expr` is a cell array of strings and `replstr` is a single string, `regexprep` uses the same replacement string on each expression in `expr`.
- If `expr` is a single string and `replstr` is a cell array of N strings, `regexprep` attempts to make N matches and replacements.
- If both `expr` and `replstr` are cell arrays of strings, then `expr` and `replstr` must contain the same number of elements, and `regexprep` pairs each `replstr` element with its matching element in `expr`. 
Examples

**Example 1 — Making a Case-Sensitive Replacement**

Perform a case-sensitive replacement on words starting with m and ending with y:

```matlab
str = 'My flowers may bloom in May';
pat = 'm(\w*)y';
regexprep(str, pat, 'April')
ans =
    My flowers April bloom in May
```

Replace all words starting with m and ending with y, regardless of case, but maintain the original case in the replacement strings:

```matlab
regexprep(str, pat, 'April', 'preserve case')
ans =
    April flowers april bloom in April
```

**Example 2 — Using Tokens In the Replacement String**

Replace all variations of the words 'walk up' using the letters following walk as a token. In the replacement string:

```matlab
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';
regexprep(str, pat, 'ascend$1')
ans =
    I ascend, they ascended, we are ascending.
```

**Example 3 — Operating on Multiple Strings**

This example operates on a cell array of strings. It searches for consecutive matching letters (e.g., 'oo') and uses a common replacement value ('--') for all matches. The function returns a cell array of strings having the same dimensions as the input cell array:

```matlab
str = {
    'Whose woods these are I think I know.' ...
    'His house is in the village though;' ...
    'He will not see me stopping here' ...
    'To watch his woods fill up with snow.'
};
```
a = regexp(str, '(.)\1', '', 'ignorecase')
a =
'Whose w--ds these are I think I know.'
'His house is in the vi--age though;'
'He wi-- not s-- me sto--ing here'
'To watch his w--ds fi-- up with snow.'

**See Also**
regexp, regexpi, regexptranslate, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi
**Purpose**
Translate string into regular expression

**Syntax**
s2 = regexptranslate(type, s1)

**Description**
s2 = regexptranslate(type, s1) translates string s1 into a regular expression string s2 that you can then use as input into one of the MATLAB regular expression functions such as regexp. The type input can be either one of the following strings that define the type of translation to be performed. See “Regular Expressions” in the MATLAB Programming Fundamentals documentation for more information.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'escape'</td>
<td>Translate all special characters (e.g., '$', '.', '?', '[') in string s1 so that they are treated as literal characters when used in the regexp and regexprep functions. The translation inserts an escape character ('') before each special character in s1. Return the new string in s2.</td>
</tr>
<tr>
<td>'wildcard'</td>
<td>Translate all wildcard and '.' characters in string s1 so that they are treated as literal wildcards and periods when used in the regexp and regexprep functions. The translation replaces all instances of '<em>' with '.</em>', all instances of '?' with '.', and all instances of '.' with '.'. Return the new string in s2.</td>
</tr>
</tbody>
</table>

**Examples**

**Example 1 — Using the ‘escape’ Option**

Because regexp interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\n' and 'n' in this string:

```matlab
str = 'The sequence \n generates a new line';
pat = '\n';

regexp(str, pat)
an =
    []
```
To have regexp interpret the expression `expr` as the characters `\` and `n`, first translate the expression using `regexptranslate`:

```matlab
pat2 = regexptranslate('escape', pat)
pat2 =
   
regexp(str, pat2)
an =
   14
```

**Example 2 — Using ‘escape’ In a Replacement String**

Replace the word 'walk' with 'ascend' in this string, treating the characters '$1' as a token designator:

```matlab
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';

regexprep(str, pat, 'ascend$1')
an =
   I ascend, they ascended, we are ascending.
```

Make another replacement on the same string, this time treating the '$1' as literal characters:

```matlab
regexprep(str, pat, regexptranslate('escape', 'ascend$1'))
an =
   I ascend$1, they ascend$1, we are ascend$1.
```

**Example 3 — Using the ‘wildcard’ Option**

Given the following string of filenames, pick out just the MAT-files. Use `regexptranslate` to interpret the '*' wildcard as '\w+' instead of as a regular expression quantifier:

```matlab
files = ['test1.mat, myfile.mat, newfile.txt, ' ...
         'jan30.mat, table3.xls'];
regexp(str, regexptranslate('wildcard', '*.mat'), 'match')
an =
```
'test1.mat' 'myfile.mat' 'jan30.mat'

To see the translation, you can type

\[
\text{regexptranslate('wildcard','*.mat')}
\]

\[
\text{ans =}
\]

\[
\text{\textbackslash w+\textbackslash .mat}
\]

\textbf{See Also}\n\text{regexp, regexpi, regexprep}
registerevent

Purpose
Register event handler for COM object event at run-time

Syntax
h.registerevent(event_handler)
registerevent(h, event_handler)

Description
h.registerevent(event_handler) registers certain event handler routines with their corresponding events. Once an event is registered, the object responds to the occurrence of that event by invoking its event handler routine. The event_handler argument can be either a string that specifies the name of the event handler function, or a function handle that maps to that function. Strings used in the event_handler argument are not case sensitive.

registerevent(h, event_handler) is an alternate syntax for the same operation.

You can either register events at the time you create the control (using actxcontrol), or register them dynamically at any time after the control has been created (using registerevent). The event_handler argument specifies both events and event handlers (see “Writing Event Handlers” in the External Interfaces documentation).

Remarks
COM functions are available on Microsoft Windows systems only.

Examples
Register Events Using Function Name Example
Create an mwsamp control and list all events associated with the control:

```matlab
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

h.events
```

MATLAB software displays (output is formatted):

```
Click = void Click()
Db1Click = void Db1Click()
MouseDown = void MouseDown(int16 Button, int16 Shift,
```
Variant x, Variant y
Event_Args = void Event_Args(int16 typeshort,
   int32 typelong, double typedouble, string typestring,
   bool typebool)

Register all events with the same event handler routine, sampev. Use eventlisteners to see the event handler used by each event:

```matlab
h.registerevent('sampev');
h.eventlisteners
```

MATLAB displays:

```
ans =
    'Click'    'sampev'
    'Db1Click' 'sampev'
    'MouseDown' 'sampev'
    'Event_Args' 'sampev'
```

Unregister these events. Now register the Click and DblClick events with the event handlers myclick and my2click, respectively. Note that the strings in the argument list are not case sensitive.

```matlab
h.unregisterallevents;
h.registerevent({'click' 'myclick'; ...
   'dblclick' 'my2click'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'click'    'myclick'
    'dblclick' 'my2click'
```

**Register Events Using Function Handle Example**

Register all events with the same event handler routine, sampev, but use a function handle (@sampev) instead of the function name:
registerevent

```matlab
h = actxcontrol('mwsamp.mwsampctrl1.2', [0 0 200 200]);
registerevent(h, @sampev);
```

**Register Workbook Events Example**

Create a Microsoft Excel Workbook object.

```matlab
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = wbs.Add;
```

Register all events with the same event handler routine, `AllEventHandler`.

```matlab
wb.registerevent('AllEventHandler')
wbl.eventlisteners
```

The MATLAB displays the list of all Workbook events, registered with `AllEventHandler`.

```matlab
ans =

    'Open'    'AllEventHandler'
    'Activate' 'AllEventHandler'
    'Deactivate' 'AllEventHandler'
    'BeforeClose' 'AllEventHandler'

  ...

See Also

`events (COM), eventlisteners, unregisterevent, unregisterallevents, isevent`
**Purpose**

Refresh function and file system path caches

**Syntax**

```plaintext
rehash
rehash path
rehash toolbox
rehash pathreset
rehash toolboxreset
rehash toolboxcache
```

**Description**

`rehash` with no arguments updates the MATLAB list of known files and classes for directories on the search path that are not in `matlabroot/toolbox`. It compares the timestamps for loaded functions against their timestamps on disk. It clears loaded functions if the files on disk are newer. All of this normally happens each time MATLAB displays the Command Window prompt. Use `rehash` with no arguments only when you run an M-file that updates a second M-file, and the calling file needs to reuse the updated version of the second M-file before the calling file has finished running.

`rehash path` performs the same updates as `rehash`, but uses a different technique for detecting the files and directories that require updates. Run `rehash path` only if you receive a warning during MATLAB startup notifying you that MATLAB could not tell if a directory has changed, and you encounter problems with MATLAB not using the most current versions of your M-files.

`rehash toolbox` performs the same updates as `rehash path`, except it updates the list of known files and classes for all directories on the search path, including those in `matlabroot/toolbox`. Run `rehash toolbox` when you change, add, or remove files in `matlabroot/toolbox` during a session. Typically, you should not make changes to files and directories in `matlabroot/toolbox`.

`rehash pathreset` performs the same updates as `rehash path`, and also ensures the known files and classes list follows precedence rules for shadowed functions.
**rehash**

`rehash toolboxreset` performs the same updates as `rehash toolbox`, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash toolboxcache` performs the same updates as `rehash toolbox`, and also updates the cache file. This is the equivalent of clicking the **Update Toolbox Path Cache** button in the General Preferences dialog box.

**See Also**

`addpath`, `clear`, `matlabroot`, `path`, `rmpath`

“Toolbox Path Caching in the MATLAB Program” and “Search Path” in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**  
Release COM interface

**Syntax**  
```matlab
h.release
release(h)
```

**Description**  
h.release releases the interface and all resources used by the interface. Each interface handle must be released when you are finished manipulating its properties and invoking its methods. Once an interface has been released, it is no longer valid. Subsequent operations on the MATLAB object that represents that interface will result in errors.

release(h) is an alternate syntax for the same operation.

**Note**  
Releasing the interface does not delete the control itself (see `delete`), since other interfaces on that object may still be active. See Releasing Interfaces in the External Interfaces documentation for more information.

**Remarks**  
COM functions are available on Microsoft Windows systems only.

**Examples**  
Create a Microsoft Calendar application. Then create a TitleFont interface and use it to change the appearance of the font of the calendar's title:

```matlab
f = figure('position',[300 300 500 500]);
cal = actxcontrol('mscal.calendar', [0 0 500 500], f);

TFont = cal.TitleFont;
TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

When you're finished working with the title font, release the TitleFont interface:

```matlab
TFont.release;
```
Now create a GridFont interface and use it to modify the size of the calendar’s date numerals:

```plaintext
GFont = cal.GridFont;
GFont.Size = 16;
```

When you’re done, delete the cal object and the figure window:

```plaintext
cal.delete;
delete(f);
clear f;
```

**See Also**
delete (COM), save (COM), load (COM), actxcontrol, actxserver
### Purpose
Equality and sorting of handle objects

### Syntax
- `TF = eq(H1,H2)`
- `TF = ne(H1,H2)`
- `TF = lt(H1,H2)`
- `TF = le(H1,H2)`
- `TF = gt(H1,H2)`
- `TF = ge(H1,H2)`

### Description
- `TF = eq(H1,H2)`
- `TF = ne(H1,H2)`
- `TF = lt(H1,H2)`
- `TF = le(H1,H2)`
- `TF = gt(H1,H2)`
- `TF = ge(H1,H2)`

For each pair of input arrays (`H1` and `H2`), a logical array of the same size is returned in which each element is an element-wise equality or comparison test result. These methods perform scalar expansion in the same way as the MATLAB built-in functions. See relationaloperators for more information.

You can make the following assumptions about the result of a handle comparison:

- The same two handles always compare as equal and the repeated comparison of any two handles always yields the same result in the same MATLAB session.
- Different handles are always not-equal.
- The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.
- If the input arrays belong to different classes (including the case where one input array belongs to a non-handle class such as `double`) then the comparison is always false.
• If a comparison is made between a handle object and an object of a dominant class, the method of the dominant class is invoked. You should generally test only like objects because a dominant class might not define one of these methods.

• An error occurs if the input arrays are not the same size and neither is scalar.

**See Also**

handle, meta.class
Purpose

Remainder after division

Syntax

R = rem(X,Y)

Description

R = rem(X,Y) if Y ≠ 0, returns X - n.*Y where n = fix(X./Y). If Y is not an integer and the quotient X./Y is within roundoff error of an integer, then n is that integer. The inputs X and Y must be real arrays of the same size, or real scalars.

The following are true by convention:

- rem(X,0) is NaN
- rem(X,X) for X ≠ 0 is 0
- rem(X,Y) for X = Y and Y ≠ 0 has the same sign as X.

Remarks

mod(X,Y) for X = Y and Y ≠ 0 has the same sign as Y.

rem(X,Y) and mod(X,Y) are equal if X and Y have the same sign, but differ by Y if X and Y have different signs.

The rem function returns a result that is between 0 and sign(X)*abs(Y). If Y is zero, rem returns NaN.

See Also

mod
**Purpose**
Remove key-value pairs from containers.Map

**Syntax**
`remove(M, keys)`

**Description**
`remove(M, keys)` erases all specified keys, and the values associated with them, from Map object `M.keys` can be a scalar key or a cell array of keys.

Using `remove` changes the count of the elements in the map.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

**Examples**
Create a Map object containing the names of several US states and the capital city of each:

```matlab
US_Capitals = containers.Map( ...
    {'Arizona', 'Nebraska', 'Nevada', 'New York', ...
    'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...
    {'Phoenix', 'Lincoln', 'Carson City', 'Albany', ...
    'Atlanta', 'Juneau', 'Montpelier', 'Salem'});
```

After checking how many keys there are in the `US_Capitals` map, remove the key-value pair with key name Oregon from it:

```matlab
US_Capitals.Count
ans =
    8
remove(US_Capitals, 'Oregon');

US_Capitals.Count
ans =
    7
```

Remove three more key-value pairs from the map:
remove(US_Capitals, {'Nebraska', 'Nevada', 'New York'});
US_Capitals.Count
ans =
  4

See Also
containers.Map, keys(Map), values(Map), size(Map), length(Map) isKey(Map), handle
removets

Purpose
Remove timeseries objects from ts_collection object.

Syntax
\( tsc = \text{removets}(tsc, \text{Name}) \)

Description
\( tsc = \text{removets}(tsc, \text{Name}) \) removes one or more timeseries objects with the name specified in Name from the tscollection object tsc. Name can either be a string or a cell array of strings.

Examples
The following example shows how to remove a time series from a ts_collection.

1 Create two timeseries objects, ts1 and ts2.

\[
\begin{align*}
\text{ts1} &= \text{timeseries}([1.1 \ 2.9 \ 3.7 \ 4.0 \ 3.0],1:5,\text{'name'},\text{'acceleration'}); \\
\text{ts2} &= \text{timeseries}([3.2 \ 4.2 \ 6.2 \ 8.5 \ 1.1],1:5,\text{'name'},\text{'speed'});
\end{align*}
\]

2 Create a ts_collection object tsc, which includes ts1 and ts2.

\[ tsc = \text{ts_collection}([\text{ts1} \ \text{ts2}]); \]

3 To view the members of tsc, type the following at the MATLAB prompt:

\[ tsc \]

The response is

Time Series Collection Object: unnamed

Time vector characteristics

Start time 1 seconds
End time 5 seconds

Member Time Series Objects:
removets

acceleration
speed

The members of tsc are listed by name at the bottom: acceleration and speed. These are the Name properties of ts1 and ts2, respectively.

4 Remove ts2 from tsc.

\[
tsc = \text{removets}(tsc, 'speed');
\]

5 To view the current members of tsc, type the following at the MATLAB prompt:

\[
tsc
\]

The response is

Time Series Collection Object: unnamed

Time vector characteristics

Start time 1 seconds
End time 5 seconds

Member Time Series Objects:
acceleration

The remaining member of tsc is acceleration. The timeseries speed has been removed.

See Also
addts, tscollection
rename

Purpose
Rename file on FTP server

Syntax
rename(f,'oldname','newname')

Description
rename(f,'oldname','newname') changes the name of the file oldname to newname in the current directory of the FTP server f, where f was created using ftp.

Examples
Connect to server testsite, view the contents, and change the name of testfile.m to showresults.m.

    test=ftp('ftp.testsite.com');
    dir(test)
        .    ..    testfile.m
    rename(test,'testfile.m','showresults.m')
    dir(test)
        .    ..    showresults.m

See Also
dir (ftp), delete (ftp), ftp, mget, mput
Purpose

Replicate and tile array

Syntax

\[ B = \text{repmat}(A,m,n) \]
\[ B = \text{repmat}(A,[m \ n]) \]
\[ B = \text{repmat}(A,[m \ n \ p \ldots]) \]

Description

\[ B = \text{repmat}(A,m,n) \] creates a large matrix \( B \) consisting of an \( m \)-by-\( n \) tiling of copies of \( A \). The size of \( B \) is \([\text{size}(A,1) \times m, \text{size}(A,2) \times n]\). The statement \( \text{repmat}(A,n) \) creates an \( n \)-by-\( n \) tiling.

\[ B = \text{repmat}(A,[m \ n]) \] accomplishes the same result as \( \text{repmat}(A,m,n) \).

\[ B = \text{repmat}(A,[m \ n \ p \ldots]) \] produces a multidimensional array \( B \) composed of copies of \( A \). The size of \( B \) is \([\text{size}(A,1) \times m, \text{size}(A,2) \times n, \text{size}(A,3) \times p, \ldots]\).

Remarks

\( \text{repmat}(A,m,n) \), when \( A \) is a scalar, produces an \( m \)-by-\( n \) matrix filled with \( A \)'s value and having \( A \)'s class. For certain values, you can achieve the same results using other functions, as shown by the following examples:

- \( \text{repmat}(\text{NaN},m,n) \) returns the same result as \( \text{NaN}(m,n) \).
- \( \text{repmat}(\text{single}(\text{inf}),m,n) \) is the same as \( \text{inf}(m,n,'\text{single}') \).
- \( \text{repmat}(\text{int8}(0),m,n) \) is the same as \( \text{zeros}(m,n,'\text{int8}') \).
- \( \text{repmat}(\text{uint32}(1),m,n) \) is the same as \( \text{ones}(m,n,'\text{uint32}') \).
- \( \text{repmat}(\text{eps},m,n) \) is the same as \( \text{eps}(\text{ones}(m,n)) \).

Examples

In this example, \( \text{repmat} \) replicates 12 copies of the second-order identity matrix, resulting in a “checkerboard” pattern.

\[ B = \text{repmat}([1 \text{ zeros}(2)]) , 3,4) \]

\[
B =
\begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\]
The statement \( N = \text{repmat}(\text{NaN}, [2 3]) \) creates a 2-by-3 matrix of NaNs.

### See Also
- `bsxfun`
- `NaN`
- `Inf`
- `ones`
- `zeros`
Purpose
Select or interpolate timeseries data using new time vector

Syntax
\[
\text{ts} = \text{resample}(	ext{ts}, \text{Time}) \\
\text{ts} = \text{resample}(	ext{ts}, \text{Time}, \text{interp\_method}) \\
\text{ts} = \text{resample}(	ext{ts}, \text{Time}, \text{interp\_method}, \text{code})
\]

Description
\[
\text{ts} = \text{resample}(	ext{ts}, \text{Time}) \text{ resamples the timeseries object ts using the new Time vector. When ts uses date strings and Time is numeric, Time is treated as specified relative to the ts.TimeInfo.StartDate property and in the same units that ts uses. The resample operation uses the default interpolation method, which you can view by using the getinterpmethod(ts) syntax.}
\]
\[
\text{ts} = \text{resample}(	ext{ts}, \text{Time}, \text{interp\_method}) \text{ resamples the timeseries object ts using the interpolation method given by the string interp\_method. Valid interpolation methods include 'linear' and 'zoh' (zero-order hold).}
\]
\[
\text{ts} = \text{resample}(	ext{ts}, \text{Time}, \text{interp\_method}, \text{code}) \text{ resamples the timeseries object ts using the interpolation method given by the string interp\_method. The integer code is a user-defined Quality code for resampling, applied to all samples.}
\]

Examples
The following example shows how to resample a timeseries object.

1 Create a timeseries object.
\[
\text{ts}=\text{timeseries}([1.1 2.9 3.7 4.0 3.0],1:5,'Name','speed');
\]

2 Transpose ts to make the data columnwise.
\[
\text{ts}=\text{transpose}(	ext{ts})
\]

The display in the MATLAB Command Window is

Time Series Object: speed
Time vector characteristics
resample (timeseries)

Length 5
Start time 1 seconds
End time 5 seconds

Data characteristics

Interpolation method linear
Size [5 1]
Data type double

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Note that the interpolation method is set to linear, by default.

3 Resample ts using its default interpolation method.

```
res_ts = resample(ts,[1 1.5 3.5 4.5 4.9])
```

The resampled time series displays as follows:

Time Series Object: speed

Time vector characteristics

Length 5
Start time 1 seconds
End time 4.900000e+000 seconds
Data characteristics

<table>
<thead>
<tr>
<th>Interpolation method</th>
<th>linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>[5 1]</td>
</tr>
<tr>
<td>Data type</td>
<td>double</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>3.85</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>3.1</td>
<td></td>
</tr>
</tbody>
</table>

See Also

getinterpmethod, setinterpmethod, synchronize, timeseries
Purpose

Select or interpolate data in \texttt{tscollection} using new time vector.

Syntax

\begin{verbatim}
tsc = resample(tsc,Time)
tsc = resample(tsc,Time,interp_method)
tsc = resample(tsc,Time,interp_method,code)
\end{verbatim}

Description

\texttt{tsc = resample(tsc,Time)} resamples the \texttt{tscollection} object \texttt{tsc} on the new \texttt{Time} vector. When \texttt{tsc} uses date strings and \texttt{Time} is numeric, \texttt{Time} is treated as numerical specified relative to the \texttt{tsc.TimeInfo.StartDate} property and in the same units that \texttt{tsc} uses. The \texttt{resample} method uses the default interpolation method for each time series member.

\texttt{tsc = resample(tsc,Time,interp_method)} resamples the \texttt{tscollection} object \texttt{tsc} using the interpolation method given by the string \texttt{interp_method}. Valid interpolation methods include \texttt{'linear'} and \texttt{'zoh'} (zero-order hold).

\texttt{tsc = resample(tsc,Time,interp_method,code)} resamples the \texttt{tscollection} object \texttt{tsc} using the interpolation method given by the string \texttt{interp_method}. The integer \texttt{code} is a user-defined quality code for resampling, applied to all samples.

Examples

The following example shows how to resample a \texttt{tscollection} that consists of two \texttt{timeseries} members.

1. Create two \texttt{timeseries} objects.

\begin{verbatim}
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
\end{verbatim}

2. Create a \texttt{tscollection} \texttt{tsc}.

\begin{verbatim}
tsc=tscollection({ts1 ts2});
\end{verbatim}

The time vector of the collection \texttt{tsc} is [1:5], which is the same as for \texttt{ts1} and \texttt{ts2} (individually).
3 Get the interpolation method for acceleration by typing

```matlab
tsc.acceleration
```

MATLAB responds with

**Time Series Object: acceleration**

**Time vector characteristics**

- Length: 5
- Start time: 1 seconds
- End time: 5 seconds

**Data characteristics**

- Interpolation method: linear
- Size: [1 1 5]
- Data type: double

4 Set the interpolation method for speed to zero-order hold by typing

```matlab
setinterpmethod(tsc.speed,'zoh')
```

MATLAB responds with

**Time Series Object: acceleration**

**Time vector characteristics**

- Length: 5
- Start time: 1 seconds
- End time: 5 seconds
Data characteristics

Interpolation method    zoh
Size                    [1 1 5]
Data type               double

5 Resample the time-series collection tsc by individually resampling each time-series member of the collection and using its interpolation method.

res_tsc=resample(tsc,[1 1.5 3.5 4.5 4.9])

See Also
getinterpmethod, setinterpmethod, ts_collection
Purpose

Reset graphics object properties to their defaults

Syntax

reset(h)

Description

reset(h) resets all properties having factory defaults on the object identified by h. To see the list of factory defaults, use the statement

    get(0,'factory')

If h is a figure, the MATLAB software does not reset Position, Units, WindowStyle, or PaperUnits. If h is an axes, MATLAB does not reset Position and Units.

Examples

reset(gca) resets the properties of the current axes.
reset(gcf) resets the properties of the current figure.

See Also

cla, clf, gca, gcf, hold

“Object Manipulation” on page 1-107 for related functions
**Purpose**
Reset random stream

**Class**
@RandStream

**Syntax**
reset(s)
reset(s,seed)

**Description**
reset(s) resets the generator for the random stream s to its initial internal state. This is similar to clearing s and recreating it using RandStream('type',...), except reset does not set the stream's RandnAlg, Antithetic, and FullPrecision properties to their original values.

reset(s,seed) resets the generator for the random stream s to the initial internal state corresponding to the seed seed. Resetting a stream's seed can invalidate independence with other streams.

**Note**
Resetting a stream should be used primarily for reproducing results.

**Examples**

1 Create a random stream object.

   ```matlab
   s=RandStream('mt19937ar')
   ```

2 Make it the default stream.

   ```matlab
   RandStream.setDefaultStream(s)
   ```

3 Reset the stream object you just created and generate 5 uniform random values using the rand method.

   ```matlab
   rand(s,1,5)
   ```

   ans =

   0.3631  0.4048  0.1490  0.9438  0.1247
4 Reset the stream.

    reset(s)

5 Generate the same 5 random values from the default stream.

    rand(s,1,5)

    ans =

        0.3631  0.4048  0.1490  0.9438  0.1247

See Also  @RandStream
Purpose
Reshape array

Syntax
B = reshape(A,m,n)
B = reshape(A,m,n,p,...)
B = reshape(A,[m n p ...])
B = reshape(A,...,[],...)
B = reshape(A,siz)

Description
B = reshape(A,m,n) returns the m-by-n matrix B whose elements are taken column-wise from A. An error results if A does not have m*n elements.

B = reshape(A,m,n,p,...) or B = reshape(A,[m n p ...]) returns an n-dimensional array with the same elements as A but reshaped to have the size m-by-n-by-p-by-... . The product of the specified dimensions, m*n*p*..., must be the same as prod(size(A)).

B = reshape(A,...,[],...) calculates the length of the dimension represented by the placeholder [], such that the product of the dimensions equals prod(size(A)). The value of prod(size(A)) must be evenly divisible by the product of the specified dimensions. You can use only one occurrence of [].

B = reshape(A,siz) returns an n-dimensional array with the same elements as A, but reshaped to siz, a vector representing the dimensions of the reshaped array. The quantity prod(siz) must be the same as prod(size(A)).

Examples
Reshape a 3-by-4 matrix into a 2-by-6 matrix.

A =
     1     4     7    10
     2     5     8    11
     3     6     9    12

B = reshape(A,2,6)

B =
B = reshape(A,2,[])

B =

1  3  5  7  9  11
2  4  6  8 10 12

See Also

shiftdim, squeeze, circshift, permute

The colon operator :
**Purpose**
Convert between partial fraction expansion and polynomial coefficients

**Syntax**
\[ [r,p,k] = \text{residue}(b,a) \]
\[ [b,a] = \text{residue}(r,p,k) \]

**Description**
The `residue` function converts a quotient of polynomials to pole-residue representation, and back again.

\[ [r,p,k] = \text{residue}(b,a) \]
finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, \( b(s) \) and \( a(s) \), of the form

\[
\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + b_3 s^{m-2} + \ldots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + a_3 s^{n-2} + \ldots + a_{n+1}}
\]

where \( b_j \) and \( a_j \) are the \( j \)th elements of the input vectors \( b \) and \( a \).

\[ [b,a] = \text{residue}(r,p,k) \]
converts the partial fraction expansion back to the polynomials with coefficients in \( b \) and \( a \).

**Definition**
If there are no multiple roots, then

\[
\frac{b(s)}{a(s)} = \frac{r_1}{s - p_1} + \frac{r_2}{s - p_2} + \ldots + \frac{r_n}{s - p_n} + k(s)
\]

The number of poles \( n \) is

\[
n = \text{length}(a) - 1 = \text{length}(r) = \text{length}(p)
\]

The direct term coefficient vector is empty if \( \text{length}(b) < \text{length}(a) \); otherwise

\[
\text{length}(k) = \text{length}(b) \cdot \text{length}(a) + 1
\]

If \( p(j) = \ldots = p(j+m-1) \) is a pole of multiplicity \( m \), then the expansion includes terms of the form
\[
\frac{r_j}{s-p_j} + \frac{r_{j+1}}{(s-p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s-p_j)^m}
\]

**Arguments**

- **b, a** Vectors that specify the coefficients of the polynomials in descending powers of \(s\)
- **r** Column vector of residues
- **p** Column vector of poles
- **k** Row vector of direct terms

**Algorithm**

It first obtains the poles with roots. Next, if the fraction is nonproper, the direct term \(k\) is found using deconv, which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed. For repeated roots, \texttt{resid} computes the residues at the repeated root locations.

**Limitations**

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, \(a(s)\), is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

**Examples**

If the ratio of two polynomials is expressed as

\[
\frac{b(s)}{a(s)} = \frac{5s^3 + 3s^2 - 2s + 7}{-4s^3 + 8s + 3}
\]

then

\[
b = [5 \ 3 \ -2 \ 7] \\
a = [-4 \ 0 \ 8 \ 3]
\]
and you can calculate the partial fraction expansion as

\[ [r, p, k] = \text{residue}(b,a) \]

\[
\begin{align*}
  r & = \\
  & = \begin{bmatrix} -1.4167 \cr -0.6653 \cr 1.3320 \end{bmatrix} \\
  p & = \\
  & = \begin{bmatrix} 1.5737 \cr -1.1644 \cr -0.4093 \end{bmatrix} \\
  k & = \\
  & = -1.2500
\end{align*}
\]

Now, convert the partial fraction expansion back to polynomial coefficients.

\[ [b,a] = \text{residue}(r,p,k) \]

\[
\begin{align*}
  b & = \\
  & = \begin{bmatrix} -1.2500 & -0.7500 & 0.5000 & -1.7500 \end{bmatrix} \\
  a & = \\
  & = \begin{bmatrix} 1.0000 & -0.0000 & -2.0000 & -0.7500 \end{bmatrix}
\end{align*}
\]

The result can be expressed as

\[
\frac{b(s)}{a(s)} = \frac{-1.25s^3 - 0.75s^2 + 0.50s - 1.75}{s^3 - 2.00s - 0.75}
\]

Note that the result is normalized for the leading coefficient in the denominator.

See Also: deconv, poly, roots
References

**Purpose**

Restore default search path

**Syntax**

```
restoredefaultpath
restoredefaultpath; matlabrc
```

**Description**

`restoredefaultpath` sets the search path to include only installed products from The MathWorks. Run `restoredefaultpath` if you are having problems with the search path. If `restoredefaultpath` seems to correct the problem, run `savepath`. Start the MATLAB program again to be sure the problem does not reappear.

`restoredefaultpath; matlabrc` sets the search path to include only installed products from The MathWorks and corrects path problems encountered during startup. Run `restoredefaultpath; matlabrc` if you are having problems with the search path and if `restoredefaultpath` by itself does not correct the problem. After the problem seems to be resolved, run `savepath`. Start MATLAB again to be sure the problem does not reappear.

**See Also**

addpath, path, rmpath, savepath

Search Path in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**
Reissue error

**Syntax**
rethrow(err)

**Description**
rethrow(err) reissues the error specified by err. The currently running M-file terminates and control returns to the keyboard (or to any enclosing catch block). The err argument must be a MATLAB structure containing at least one of the following fields.

<table>
<thead>
<tr>
<th>Fieldname</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>Text of the error message</td>
</tr>
<tr>
<td>identifier</td>
<td>Message identifier of the error message</td>
</tr>
<tr>
<td>stack</td>
<td>Information about the error from the program stack</td>
</tr>
</tbody>
</table>

See "Message Identifiers" in the MATLAB documentation for more information on the syntax and usage of message identifiers.

A convenient way to get a valid err structure for the last error issued is by using the lasterror function.

**Remarks**
The err input can contain the field stack, identical in format to the output of the dbstack command. If the stack field is present, the stack of the rethrown error will be set to that value. Otherwise, the stack will be set to the line at which the rethrow occurs.

**Examples**
rethrow is usually used in conjunction with try-catch statements to reissue an error from a catch block after performing catch-related operations. For example,

```
try
do_something
catch
do_cleanup
  rethrow(lasterror)
end
```
rethrow

See Also  error, lasterror, try, catch, dbstop
Purpose

Reissue existing exception

Syntax

rethrow(ME)

Description

rethrow(ME) terminates the currently running function, reissues an exception that is based on MException object ME that has been caught within a try-catch block, and returns control to the keyboard or to any enclosing catch block.

rethrow differs from the throw and throwAsCaller methods in that it does not modify the stack field. Call stack information in the ME object is kept as it was when the exception was first thrown.

rethrow can only issue a previously caught exception. If an exception that was not previously thrown is passed to the rethrow method, the MATLAB software generates a new exception.

You might use rethrow from the catch part of a try-catch block, for example, after performing some required cleanup tasks following an error.

Examples

This variation of the MATLAB surf function catches an error in the input arguments, gives the user the opportunity to correct the error, and rethrows the error if the user does not use that opportunity:

```matlab
function surf2(varargin)
    try
        surf(varargin{:})
    catch ME
        ME.message % Display the error.
        % Give user another try to enter input arguments.
        newargs = input('
Enter argument list: ','s');
        if ~isempty(newargs)
            surf(eval(newargs));
        else
            % If no response from user, rethrow the error.
            rethrow(ME);
        end
    end
```
When asked to correct the error, the user presses **Enter**. MATLAB rethrows the original error:

```matlab
surf2
ans =
Not enough input arguments.

Enter argument list:
??? Error using ==> surf at 54
Not enough input arguments.

Error in ==> surf2 at 3
    surf(varargin{:});
```

This time, the user enters valid input and MATLAB successfully displays the output plot:

```matlab
surf2
ans =
Not enough input arguments.

Enter argument list: peaks(30)
```

**See Also**

`try`, `catch`, `error`, `assert`, `MException`, `throw(MException)`, `throwAsCaller(MException)`, `addCause(MException)`, `getReport(MException)`, `disp(MException)`, `isequal(MException)`, `eq(MException)`, `ne(MException)`, `last(MException)`
**Purpose**
Return to invoking function

**Syntax**
return

**Description**
return causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.

**Examples**
If the determinant function were an M-file, it might use a return statement in handling the special case of an empty matrix, as follows:

```matlab
function d = det(A)
    %DET det(A) is the determinant of A.
    if isempty(A)
        d = 1;
        return
    else

        ...
    end
```

**See Also**
break, continue, disp, end, error, for, if, keyboard, switch, while
Purpose
Convert RGB colormap to HSV colormap

Syntax
```
cmap = rgb2hsv(M)
hsv_image = rgb2hsv(rgb_image)
```

Description
cmap = rgb2hsv(M) converts an RGB colormap M to an HSV colormap cmap. Both colormaps are m-by-3 matrices. The elements of both colormaps are in the range 0 to 1.

The columns of the input matrix M represent intensities of red, green, and blue, respectively. The columns of the output matrix cmap represent hue, saturation, and value, respectively.

hsv_image = rgb2hsv(rgb_image) converts the RGB image to the equivalent HSV image. RGB is an m-by-n-by-3 image array whose three planes contain the red, green, and blue components for the image. HSV is returned as an m-by-n-by-3 image array whose three planes contain the hue, saturation, and value components for the image.

See Also
brighten, colormap, hsv2rgb, rgbplot

“Color Operations” on page 1-105 for related functions
Purpose

Convert RGB image to indexed image

Syntax

\[
\begin{align*}
[X, \text{map}] &= \text{rgb2ind}(\text{RGB}, \ n) \\
X &= \text{rgb2ind}(\text{RGB}, \ \text{map}) \\
[X, \text{map}] &= \text{rgb2ind}(\text{RGB}, \ \text{tol}) \\
[\ldots] &= \text{rgb2ind}(\ldots, \ \text{dither\_option})
\end{align*}
\]

Description

\text{rgb2ind} converts RGB images to indexed images using one of these methods:

- Uniform quantization
- Minimum variance quantization
- Colormap approximation

For all these methods, \text{rgb2ind} also dithers the image unless you specify 'nodither' for \text{dither\_option}.

\[X, \text{map}] = \text{rgb2ind}(\text{RGB}, \ n)\] converts the RGB image to an indexed image \(X\) using minimum variance quantization. \text{map} contains at most \(n\) colors. \(n\) must be less than or equal to 65,536.

\(X = \text{rgb2ind}(\text{RGB}, \ \text{map})\) converts the RGB image to an indexed image \(X\) with colormap \text{map} by matching colors in \text{RGB} with the nearest color in the colormap \text{map}. size(\text{map},1) must be less than or equal to 65,536.

\[X, \text{map}] = \text{rgb2ind}(\text{RGB}, \ \text{tol})\] converts the RGB image to an indexed image \(X\) using uniform quantization. \text{map} contains at most \((\text{floor}(1/\text{tol})+1)^3\) colors. \text{tol} must be between 0.0 and 1.0.

\[\ldots\] = \text{rgb2ind}(\ldots, \ \text{dither\_option})\] enables or disables dithering. \text{dither\_option} is a string that can have one of these values.
'dither' (default) dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.

'nodither' maps each color in the original image to the closest color in the new map. No dithering is performed.

**Note** The values in the resultant image X are indexes into the colormap map and cannot be used in mathematical processing, such as filtering operations.

**Class Support**

The input image can be of class `uint8`, `uint16`, `single`, or `double`. If the length of map is less than or equal to 256, the output image is of class `uint8`. Otherwise, the output image is of class `uint16`.

**Remarks**

If you specify `tol`, `rgb2ind` uses uniform quantization to convert the image. This method involves cutting the RGB color cube into smaller cubes of length `tol`. For example, if you specify a `tol` of 0.1, the edges of the cubes are one-tenth the length of the RGB cube. The total number of small cubes is:

\[ n = \left( \text{floor}(1/tol) + 1 \right)^3 \]

Each cube represents a single color in the output image. Therefore, the maximum length of the colormap is \( n \). `rgb2ind` removes any colors that don't appear in the input image, so the actual colormap can be much smaller than \( n \).

If you specify `n`, `rgb2ind` uses minimum variance quantization. This method involves cutting the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number you specify, the output colormap is also smaller.
If you specify map, rgb2ind uses colormap mapping, which involves finding the colors in map that best match the colors in the RGB image.

**Examples**

```matlab
RGB = imread('peppers.png');
[X,map] = rgb2ind(RGB,128);
figure, imshow(X,map)
```

**See Also**

cmunique, dither, imapprox, ind2rgb
**Purpose**
Plot colormap

```
<table>
<thead>
<tr>
<th>0</th>
<th>0.5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Syntax**
```
rgbplot(cmap)
```

**Description**
`rgbplot(cmap)` plots the three columns of `cmap`, where `cmap` is an `m`-by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

**Examples**
Plot the RGB values of the `copper` colormap.
```
rgbplot(copper)
```
See Also  

colormap

“Color Operations” on page 1-105 for related functions
Purpose

Ribbon plot

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
ribbon(axes_handle,...)
h = ribbon(...)

Description

ribbon(Y) plots the columns of Y as undulating three-dimensional ribbons of uniform width using \( X = 1: \text{size}(Y,1) \). Ribbons advance along the x-axis centered on tick marks at unit intervals, three-quarters of a unit in width. Ribbons are assigned colors from the current colormap in sequence from minimum X to maximum X (the axes colororder property, used by plot and plot3, does not apply to ribbon or other surface plots).

ribbon(X,Y) plots X versus the columns of Y as three-dimensional strips. X and Y are vectors of the same size or matrices of the same size. Additionally, X can be a row or a column vector, and Y a matrix with length(X) rows. ribbon(X,Y) is the same as plot(X,Y) except that the columns of Y are plotted as separated ribbons in 3-D. The y and z-axes of ribbon(X,Y) correspond to the x and y-axes of plot(X,Y).

ribbon(X,Y,width) specifies the width of the ribbons. The default is 0.75. If width = 1, the ribbons touch, leaving no space between them when viewed down the z-axis. If width > 1, ribbons overlap and can intersect.
ribbon(axes_handle, ...) plots into the axes with handle axes_handle instead of the current axes (gca).

h = ribbon(...) returns a vector of handles to surface graphics objects. ribbon returns one handle per strip.

Examples

Create a ribbon plot of the peaks function.

```matlab
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
z = peaks(x,y);
ribbon(y,z)
xlabel('X')
ylabel('Y')
zlabel('Z')
colormap hsv
```
ribbon

See Also  
plot, plot3, surface, waterfall

“Polygons and Surfaces” on page 1-97 for related functions
| **Purpose** | Remove application-defined data |
| **Syntax** | `rmappdata(h,name)` |
| **Description** | `rmappdata(h,name)` removes the application-defined data `name` from the object specified by handle `h`. |
| **Remarks** | Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property. |
| **See Also** | `getappdata`, `isappdata`, `setappdata` |
rmdir

Purpose
Remove directory

Graphical Interface
As an alternative to the rmdir function, use the delete feature in the Current Directory browser.

Syntax
rmdir('dirname')

rmdir('dirname','s')

[status, message, messageid] = rmdir('dirname','s')

Description
rmdir('dirname') removes the directory dirname from the current directory. If the directory is not empty, you must use the 's' argument. If dirname is not in the current directory, specify the relative path to the current directory or the full path for dirname.

rmdir('dirname','s') removes the directory dirname and its contents from the current directory. This removes all subdirectories and files in the current directory regardless of their write permissions.

[status, message, messageid] = rmdir('dirname','s') removes the directory dirname and its contents from the current directory, returning the status, a message, and the MATLAB error message ID (see error and lasterror). Here, status is 1 for success and is 0 for error, and message, messageid, and the s input argument are optional.

Remarks
When attempting to remove multiple directories, either by including a wildcard in the directory name or by specifying the 's' flag in the rmdir command, MATLAB throws an error if it is unable to remove all directories to which the command applies. The error message contains a listing of those directories and files that MATLAB could not remove.

Examples
Remove Empty Directory
To remove myfiles from the current directory, where myfiles is empty, type

rmdir('myfiles')
If the current directory is `matlabr13/work`, and `myfiles` is in `d:/matlabr13/work/project/`, use the relative path to `myfiles`:

```
rmdir('project/myfiles')
```
or the full path to `myfiles`:

```
rmdir('d:/matlabr13/work/project/myfiles')
```

**Remove Directory and All Contents**

To remove `myfiles`, its subdirectories, and all files in the directories, assuming `myfiles` is in the current directory, type

```
rmdir('myfiles','s')
```

**Remove Directory and Return Results**

To remove `myfiles` from the current directory, type

```
[stat, mess, id]=rmdir('myfiles')
```

MATLAB returns

```
stat =
  0

mess =

The directory is not empty.

id =

MATLAB:RMDIR:OSErr
```

indicating the directory `myfiles` is not empty.

To remove `myfiles` and its contents, run

```
[stat, mess]=rmdir('myfiles','s')
```

and MATLAB returns
stat =
1

mess =
.

indicating myfiles and its contents have been removed.

See Also

cd, copyfile, delete, dir, error, fileattrib, filebrowser, lasterror, mkdir, movefile

“Managing Files and Working with the Current Directory”
Purpose
Remove directory on FTP server

Syntax
rmdir(f,'dirname')

Description
rmdir(f,'dirname') removes the directory dirname from the current directory of the FTP server f, where f was created using ftp.

Examples
Connect to server testsite, view the contents of testdir, and remove the directory newdir from the directory testdir.

```matlab
test=ftp('ftp.testsite.com');
cd(test,'testdir');
dir(test)
    .    .    newdir
dir(test,'newdir')
    .    .
rmrdir(test,'newdir');
dir(test,'testdir')
    .    .
```

See Also
cd (ftp), delete (ftp), dir (ftp), ftp, mkdir (ftp)
Purpose
Remove fields from structure

Syntax
s = rmfield(s, 'fieldname')
s = rmfield(s, fields)

Description
s = rmfield(s, 'fieldname') removes the specified field from the
structure array s.

s = rmfield(s, fields) removes more than one field at a time.
fields is a character array of field names or cell array of strings.

See Also
fieldnames, setfield, getfield, isfield, orderfields, dynamic
field names
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Remove directories from search path</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GUI</strong></td>
<td>As an alternative to the <code>rmpath</code> function, use the Set Path dialog box.</td>
</tr>
<tr>
<td><strong>Alternatives</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td><code>rmpath('directory')</code></td>
</tr>
<tr>
<td></td>
<td><code>rmpath directory</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>rmpath('directory')</code> removes the specified directory from the current MATLAB search path. Use the full pathname for <code>directory</code>. <code>rmpath directory</code> is the command form of the syntax.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>Remove <code>/usr/local/matlab/mytools</code> from the search path.</td>
</tr>
<tr>
<td></td>
<td><code>rmpath /usr/local/matlab/mytools</code></td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td><code>addpath</code>, <code>cd</code>, <code>dir</code>, <code>genpath</code>, <code>matlabroot</code>, <code>partialpath</code>, <code>path</code>, <code>paths</code>, <code>pathtool</code>, <code>rehash</code>, <code>restoredefaultpath</code>, <code>savepath</code>, <code>userpath</code>, <code>what</code></td>
</tr>
<tr>
<td></td>
<td>Search Path in the MATLAB Desktop Tools and Development Environment documentation</td>
</tr>
</tbody>
</table>
Purpose

Remove preference

Syntax

\[ \text{rmpref('group','pref')} \]
\[ \text{rmpref('group',{\text{'pref1','pref2',...','prefn'}})} \]
\[ \text{rmpref('group')} \]

Description

\[ \text{rmpref('group','pref')} \] removes the preference specified by group and pref. It is an error to remove a preference that does not exist.

\[ \text{rmpref('group',{\text{'pref1','pref2',...','prefn'}})} \] removes each preference specified in the cell array of preference names. It is an error if any of the preferences do not exist.

\[ \text{rmpref('group')} \] removes all the preferences for the specified group. It is an error to remove a group that does not exist.

Examples

\[ \text{addpref('mytoolbox','version','1.0')} \]
\[ \text{rmpref('mytoolbox')} \]

See Also

addpref, getpref, ispref, setpref, uigetpref, uisetpref
**Purpose**

Root

**Description**

The root is a graphics object that corresponds to the computer screen. There is only one root object and it has no parent. The children of the root object are figures.

The root object exists when you start MATLAB; you never have to create it and you cannot destroy it. Use `set` and `get` to access the root properties.

**See Also**

diary, echo, figure, format, gcf, get, set

**Object Hierarchy**

```
  Root
    Figure
      Axes
      Uiobjects
```
**Root Properties**

**Purpose**
Root properties

**Modifying Properties**
You can set and query graphics object properties in two ways:

- The “The Property Editor” is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see “Setting Default Property Values”.

**Root Properties**
This section lists property names along with the type of values each accepts. Curly braces {} enclose default values.

- **BusyAction**
  
  cancel | {queue}

  Not used by the root object.

- **ButtonDownFcn**
  
  string

  Not used by the root object.

- **CallbackObject**
  
  handle (read only)

  *Handle of current callback’s object.* This property contains the handle of the object whose callback routine is currently executing. If no callback routines are executing, this property contains the empty matrix []). See also the `gco` command.

- **Children**
  
  vector of handles

  *Handles of child objects.* A vector containing the handles of all nonhidden figure objects (see `HandleVisibility` for more
information). You can change the order of the handles and thereby change the stacking order of the figures on the display.

Clipping

{on} | off

Clipping has no effect on the root object.

CommandWindowSize

[columns rows]

*Current size of command window.* This property contains the size of the MATLAB command window in a two-element vector. The first element is the number of columns wide and the second element is the number of rows tall.

CreateFcn

The root does not use this property.

CurrentFigure

figure handle

*Handle of the current figure window,* which is the one most recently created, clicked in, or made current with the statement

\[
\text{figure(h)}
\]

which restacks the figure to the top of the screen, or

\[
\text{set(0,'CurrentFigure',h)}
\]

which does not restack the figures. In these statements, \( h \) is the handle of an existing figure. If there are no figure objects,

\[
\text{get(0,'CurrentFigure')}
\]

returns the empty matrix. Note, however, that \( \text{gcf} \) always returns a figure handle, and creates one if there are no figure objects.

DeleteFcn

string
This property is not used, because you cannot delete the root object.

**Diary**

```
on | {off}
```

*Diary file mode.* When this property is on, MATLAB maintains a file (whose name is specified by the DiaryFile property) that saves a copy of all keyboard input and most of the resulting output. See also the `diary` command.

**DiaryFile**

```
string
```

*Diary filename.* The name of the diary file. The default name is `diary`.

**Echo**

```
on | {off}
```

*Script echoing mode.* When Echo is on, MATLAB displays each line of a script file as it executes. See also the `echo` command.

**ErrorMessage**

```
string
```

*Text of last error message.* This property contains the last error message issued by MATLAB.

**FixedWidthFontName**

```
font name
```

*Fixed-width font to use for axes, text, and uicontrols whose FontName is set to FixedWidth.* MATLAB uses the font name specified for this property as the value for axes, text, and uicontrol FontName properties when their FontName property is set to FixedWidth. Specifying the font name with this property eliminates the need to hardcode font names in MATLAB applications and thereby enables these applications to run...
without modification in locales where non-ASCII character sets are required. In these cases, MATLAB attempts to set the value of `FixedWidthFontName` to the correct value for a given locale.

MATLAB application developers should not change this property, but should create axes, text, and uicontrols with `FontName` properties set to `FixedWidth` when they want to use a fixed-width font for these objects.

MATLAB end users can set this property if they do not want to use the preselected value. In locales where Latin-based characters are used, Courier is the default.

**Format**

```
short | {shortE} | long | longE | bank | hex | + | rat
```

*Output format mode.* This property sets the format used to display numbers. See also the `format` command.

- **short** — Fixed-point format with 5 digits
- **shortE** — Floating-point format with 5 digits
- **shortG** — Fixed- or floating-point format displaying as many significant figures as possible with 5 digits
- **long** — Scaled fixed-point format with 15 digits
- **longE** — Floating-point format with 15 digits
- **longG** — Fixed- or floating-point format displaying as many significant figures as possible with 15 digits
- **bank** — Fixed-format of dollars and cents
- **hex** — Hexadecimal format
- **+** — Displays + and – symbols
- **rat** — Approximation by ratio of small integers
Root Properties

FormatSpacing
  compact | {loose}

  *Output format spacing (see also format command).*
  *compact* — Suppress extra line feeds for more compact display.
  *loose* — Display extra line feeds for a more readable display.

HandleVisibility
  {on} | callback | off

  This property is not useful on the root object.

HitTest
  {on} | off

  This property is not useful on the root object.

Interruptible
  {on} | off

  This property is not useful on the root object.

Language
  string

  System environment setting.

MonitorPosition
  [x y width height;x y width height]

  *Width and height of primary and secondary monitors, in pixels.*
  This property contains the width and height of each monitor connected to your computer. The x and y values for the primary monitor are 0, 0 and the width and height of the monitor are specified in pixels.

  The secondary monitor position is specified as

  \[ x = \text{primary monitor width} + 1 \]
\[ y = \text{primary monitor height} + 1 \]

Querying the value of the figure `MonitorPosition` on a multiheaded system returns the position for each monitor on a separate line.

```matlab
v = get(0,'MonitorPosition')
v =
x y width height % Primary monitor
x y width height % Secondary monitor
```

The value of the `ScreenSize` property is inconsistent when using multiple monitors. If you want specific and consistent values, use the `MonitorPosition` property.

**Parent**

`handle`

*Handle of parent object.* This property always contains the empty matrix, because the root object has no parent.

**PointerLocation**

`[x,y]`

*Current location of pointer.* A vector containing the \( x \)- and \( y \)-coordinates of the pointer position, measured from the lower left corner of the screen. You can move the pointer by changing the values of this property. The `Units` property determines the units of this measurement.

This property always contains the current pointer location, even if the pointer is not in a MATLAB window. A callback routine querying the `PointerLocation` can get a value different from the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.
On Macintosh platforms, you cannot change the pointer location using the `set` command.

**PointerWindow**

handle (read only)

*Handle of window containing the pointer.* MATLAB sets this property to the handle of the figure window containing the pointer. If the pointer is not in a MATLAB window, the value of this property is 0. A callback routine querying the `PointerWindow` can get the wrong window handle if you move the pointer to another window before the callback executes. This error results from delays in callback execution caused by competition for system resources.

**RecursionLimit**

integer

*Number of nested M-file calls.* This property sets a limit to the number of nested calls to M-files MATLAB will make before stopping (or potentially running out of memory). By default the value is set to a large value. Setting this property to a smaller value (something like 150, for example) should prevent MATLAB from running out of memory and will instead cause MATLAB to issue an error when the limit is reached.

**ScreenDepth**

bits per pixel

*Screen depth.* The depth of the display bitmap (i.e., the number of bits per pixel). The maximum number of simultaneously displayed colors on the current graphics device is $2$ raised to this power.

`ScreenDepth` supersedes the `BlackAndWhite` property. To override automatic hardware checking, set this property to 1. This value causes MATLAB to assume the display is monochrome. This is useful if MATLAB is running on color hardware but is
being displayed on a monochrome terminal. Such a situation can cause MATLAB to determine erroneously that the display is color.

ScreenPixelsPerInch

Display resolution

*DPI setting for your display.* This property contains the setting of your display resolution specified in your system preferences.

ScreenSize

four-element rectangle vector (read only)

*Screen size.* A four-element vector,

\[ \text{[left, bottom, width, height]} \]

that defines the display size. left and bottom are 0 for all Units except pixels, in which case left and bottom are 1. width and height are the screen dimensions in units specified by the Units property.

**Determining Screen Size**

Note that the screen size in absolute units (e.g., inches) is determined by dividing the number of pixels in width and height by the screen DPI (see the ScreenPixelsPerInch property). This value is approximate and might not represent the actual size of the screen.

Note that the ScreenSize property is static. Its values are read only at MATLAB startup and not updated if system display settings change. Also, the values returned might not represent the usable screen size for application developers due to the presence of other GUIs, such as the Microsoft Windows task bar.

Selected

on | off

This property has no effect on the root level.
**SelectionHighlight**

{on} | off

This property has no effect on the root level.

**ShowHiddenHandles**

on | {off}

*Show or hide handles marked as hidden.* When set to on, this property disables handle hiding and exposes all object handles regardless of the setting of an object’s HandleVisibility property. When set to off, all objects so marked remain hidden within the graphics hierarchy.

**Tag**

string

*User-specified object label.* The Tag property provides a means to identify graphics objects with a user-specified label. While it is not necessary to identify the root object with a tag (since its handle is always 0), you can use this property to store any string value that you can later retrieve using set.

**Type**

string (read only)

Class of graphics object. For the root object, Type is always 'root'.

**UIContextMenu**

handle

This property has no effect on the root level.

**Units**

{pixels} | normalized | inches | centimeters | points | characters
Unit of measurement. This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower left corner of the screen. Normalized units map the lower left corner of the screen to (0,0) and the upper right corner to (1.0,1.0). inches, centimeters, and points are absolute units (one point equals 1/72 of an inch). Characters are units defined by characters from the default system font; the width of one unit is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

This property affects the PointerLocation and ScreenSize properties. If you change the value of Units, it is good practice to return it to its default value after completing your operation, so as not to affect other functions that assume Units is set to the default value.

UserData

matrix

User-specified data. This property can be any data you want to associate with the root object. MATLAB does not use this property, but you can access it using the set and get functions.

Visible

{on} | off

Object visibility. This property has no effect on the root object.
**Purpose**  
Polynomial roots

**Syntax**  
r = roots(c)

**Description**  
r = roots(c) returns a column vector whose elements are the roots of the polynomial c.

Row vector c contains the coefficients of a polynomial, ordered in descending powers. If c has n+1 components, the polynomial it represents is $c_1s^n + \ldots + c_n s + c_{n+1}$.

**Remarks**  
Note the relationship of this function to p = poly(r), which returns a row vector whose elements are the coefficients of the polynomial. For vectors, roots and poly are inverse functions of each other, up to ordering, scaling, and roundoff error.

**Examples**  
The polynomial $s^3 - 6s^2 - 72s - 27$ is represented in MATLAB software as

```
p = [1 -6 -72 -27]
```

The roots of this polynomial are returned in a column vector by

```
r = roots(p)
r =
  12.1229
  -5.7345
  -0.3884
```

**Algorithm**  
The algorithm simply involves computing the eigenvalues of the companion matrix:

```
A = diag(ones(n-1,1),-1);
A(1,:) = -c(2:n+1)/c(1);
eig(A)
```
It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix $A$, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in $c$.

**See Also**

fzero, poly, residue
### Purpose
Angle histogram plot

![Angle histogram plot](image)

### GUI Alternatives
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in `plot edit` mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

### Syntax

```matlab
rose(theta)
rose(theta,x)
rose(theta,nbins)
rose(axes_handle,...)
```

```matlab
h = rose(...) 
[tout,rout] = rose(...) 
```

### Description
`rose(theta)` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range, showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle of each bin from the origin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0, 2*pi]`. The default is 20.

`rose(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

2-3156
h = rose(...) returns the handles of the line objects used to create the graph.

[tout,rout] = rose(...) returns the vectors tout and rout so polar(tout,rout) generates the histogram for the data. This syntax does not generate a plot.

Example

Create a rose plot showing the distribution of 50 random numbers.

```matlab
theta = 2*pi*rand(1,50);
rose(theta)
```
See Also

compass, feather, hist, line, polar

“Histograms” on page 1-97 for related functions

Histograms in Polar Coordinates for another example
Purpose

Classic symmetric eigenvalue test problem

Syntax

A = rosser

Description

A = rosser returns the Rosser matrix. This matrix was a challenge for many matrix eigenvalue algorithms. But LAPACK’s DSYEV routine used in MATLAB software has no trouble with it. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

Examples

rosser

ans =

611 196 -192 407 -8 -52 -49 29
196 899 113 -192 -71 -43 -8 -44
-192 113 899 196 61 49 8 52
407 -192 196 611 8 44 59 -23
-8 -71 61 8 411 599 208 208
-52 -43 49 44 599 411 208 208
-52 -8 8 59 208 208 99 -911
29 -44 52 -23 208 208 -911 99
Purpose

Rotate matrix 90 degrees

Syntax

B = rot90(A)
B = rot90(A,k)

Description

B = rot90(A) rotates matrix A counterclockwise by 90 degrees.

B = rot90(A,k) rotates matrix A counterclockwise by k*90 degrees, where k is an integer.

Examples

The matrix

\[
X =
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

rotated by 90 degrees is

\[
Y = \text{rot90}(X)
Y =
\begin{bmatrix}
3 & 6 & 9 \\
2 & 5 & 8 \\
1 & 4 & 7
\end{bmatrix}
\]

See Also

flipdim, fliplr, flipud
**Purpose**

Rotate object in specified direction

**Syntax**

```plaintext
rotate(h,direction,alpha)
rotate(...,origin)
```

**Description**

The `rotate` function rotates a graphics object in three-dimensional space, according to the right-hand rule.

`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector. The default origin is the center of the plot box.

**Remarks**

The graphics object you want rotated must be a child of the same axes. The object’s data is modified by the rotation transformation. This is in contrast to `view` and `rotate3d`, which only modify the viewpoint.

The axis of rotation is defined by an origin and a point `P` relative to the origin. `P` is expressed as the spherical coordinates `[theta, phi]` or as Cartesian coordinates.
The two-element form for direction specifies the axis direction using the spherical coordinates \([\theta \ \phi]\). \(\theta\) is the angle in the \(x-y\) plane counterclockwise from the positive \(x\)-axis. \(\phi\) is the elevation of the direction vector from the \(x-y\) plane.

The three-element form for direction specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin to \((X,Y,Z)\).

**Examples**

Rotate a graphics object 180° about the \(x\)-axis.

```matlab
h = surf(peaks(20));
rotate(h,[1 0 0],180)
```

Rotate a surface graphics object 45° about its center in the \(z\) direction.

```matlab
h = surf(peaks(20));
zdir = [0 0 1];
center = [10 10 0];
rotate(h,zdir,45,center)
```
**Remarks**
rotate changes the Xdata, Ydata, and Zdata properties of the appropriate graphics object.

**See Also**
rotate3d, sph2cart, view

The axes CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle

“Object Manipulation” on page 1-107 for related functions
**Purpose**

Rotate 3-D view using mouse

**GUI Alternatives**

Use the Rotate3D tool on the figure toolbar to enable and disable rotate3D mode on a plot, or select **Rotate 3D** from the figure’s **Tools** menu. For details, see “Rotate 3D — Interactive Rotation of 3-D Views” in the MATLAB Graphics documentation.

**Syntax**

rotate3d on  
rotate3d off  
rotate3d  
rotate3d(figure_handle,...)  
rotate3d(axes_handle,...)  
h = rotate3d(figure_handle)

**Description**

- `rotate3d on` enables mouse-base rotation on all axes within the current figure.
- `rotate3d off` disables interactive axes rotation in the current figure.
- `rotate3d` toggles interactive axes rotation in the current figure.
- `rotate3d(figure_handle,...)` enables rotation within the specified figure instead of the current figure.
- `rotate3d(axes_handle,...)` enables rotation only in the specified axes.
- `h = rotate3d(figure_handle)` returns a rotate3d mode object for figure figure_handle for you to customize the mode’s behavior.

**Using Rotate Mode Objects**

You access the following properties of rotate mode objects via `get` and modify some of them using `set`.

- **FigureHandle** `<handle>` — The associated figure handle, a read-only property that cannot be set
- **Enable** `'on'`|`'off'` — Specifies whether this figure mode is currently enabled on the figure
• **RotateStyle** 'orbit'|'box' — Sets the method of rotation

  'orbit' rotates the entire axes; 'box' rotates a plot-box outline of the axes.

### Rotate3D Mode Callbacks

You can program the following callbacks for rotate3d mode operations.

- **ButtonDownFilter** &lt;function_handle&gt; — Function to intercept ButtonDown events

  The application can inhibit the rotate operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

  ```matlab
  function [res] = myfunction(obj,event_obj)
  % obj handle to the object that has been clicked on
  % event_obj handle to event data object (empty in this release)
  % res [output] logical flag to determine whether the rotate operation should take place or the 'ButtonDownFcn' property of the object should take precedence
  ```

- **ActionPreCallback** &lt;function_handle&gt; — Function to execute before rotating

  Set this callback to listen to when a rotate operation will start. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```matlab
  function myfunction(obj,event_obj)
  % obj handle to the figure that has been clicked on
  % event_obj object containing struct of event data
  ```

  The event data has the following field:

  | Axes | The handle of the axes that is being panned |
- **ActionPostCallback <function_handle>** — Function to execute after rotating

  Set this callback to listen to when a rotate operation has finished. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj          handle to the figure that has been clicked on
  % event_obj    object containing struct of event data (same as the
  %               event data of the 'ActionPreCallback' callback)
  ```

### Rotate3D Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- **flags = isAllowAxesRotate(h,axes)** — Function querying permission to rotate axes

  Calling the function `isAllowAxesRotate` on the `rotate3d` object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a rotate operation is permitted on the axes objects.

- **setAllowAxesRotate(h,axes,flag)** — Function to set permission to pan axes

  Calling the function `setAllowAxesRotate` on the `rotate3d` object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a rotate operation on the axes objects.

### Examples

#### Example 1

Simple 3-D rotation

```matlab
surf(peaks);
rotate3d on
% rotate the plot using the mouse pointer.
```
**Example 2**

Rotate the plot using the "Plot Box" rotate style:

```matlab
surf(peaks);
h = rotate3d;
set(h,'RotateStyle','box','Enable','on');
% Rotate the plot.
```

**Example 3**

Create two axes as subplots and then prevent one from rotating:

```matlab
ax1 = subplot(1,2,1);
surf(peaks);
h = rotate3d;
ax2 = subplot(1,2,2);
surf(membrane);
setAllowAxesRotate(h,ax2,false);
% rotate the plots.
```

**Example 4**

Create a buttonDown callback for rotate mode objects to trigger. Copy the following code to a new M-file, execute it, and observe rotation behavior:

```matlab
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = rotate3d;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse-click on the line
```
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
    flag = true;
else
    flag = false;
end

Example 5
Create callbacks for pre- and post-buttonDown events for rotate3D mode objects to trigger. Copy the following code to a new M-file, execute it, and observe rotation behavior:

function demo
% Listen to rotate events
surf(peaks);
  h = rotate3d;
  set(h,'ActionPreCallback',@myprecallback);
  set(h,'ActionPostCallback',@mypostcallback);
  set(h,'Enable','on');
%
function myprecallback(obj,evd)
  disp('A rotation is about to occur.');
%
function mypostcallback(obj,evd)
  newView = round(get(evd.Axes,'View'));
  msgbox(sprintf('The new view is [%d %d].',newView));

Remarks
When enabled, rotate3d provides continuous rotation of axes and the objects it contains through mouse movement. A numeric readout appears in the lower left corner of the figure during rotation, showing the current azimuth and elevation of the axes. Releasing the mouse button removes the animated box and the readout. This differs from the camorbit function in that while the rotate3d tool modifies the View property of the axes, the camaorbit function fixes the aspect ratio
and modifies the CameraTarget, CameraPosition and CameraUpVector
properties of the axes. See Axes Properties for more information.

You can also enable 3-D rotation from the figure Tools menu or the
figure toolbar.

You can create a rotate3D mode object once and use it to customize the
behavior of different axes, as example 3 illustrates. You can also change
its callback functions on the fly.

**Note** Do not change figure callbacks within an interactive
mode. While a mode is active (when panning, zooming, etc.), you will
receive a warning if you attempt to change any of the figure’s callbacks
and the operation will not succeed. The one exception to this rule is the
figure WindowButtonMotionFcn callback, which can be changed from
within a mode. Therefore, if you are creating a GUI that updates a
figure’s callbacks, the GUI should some keep track of which interactive
mode is active, if any, before attempting to do this.

When you assign different 3-D rotation behaviors to different subplot
axes via a mode object and then link them using the linkaxes function,
the behavior of the axes you manipulate with the mouse will carry over
to the linked axes, regardless of the behavior you previously set for
the other axes.

**See Also**
camorbit, pan, rotate, view, zoom

Object Manipulation for related functions

Axes Properties for related properties
**Purpose**
Round to nearest integer

**Syntax**
Y = round(X)

**Description**
Y = round(X) rounds the elements of X to the nearest integers. For complex X, the imaginary and real parts are rounded independently.

**Examples**
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

```
a =  
Columns 1 through 4
-1.9000  -0.2000   3.4000   5.6000
Columns 5 through 6
   7.0000   2.4000 + 3.6000i
```

round(a)

```
ans =  
Columns 1 through 4
-2.0000  0.0000   3.0000   6.0000
Columns 5 through 6
   7.0000   2.0000 + 4.0000i
```

**See Also**
ceil, fix, floor
Purpose
Reduced row echelon form

Syntax
\[
\begin{align*}
R &= \text{rref}(A) \\
[R,jb] &= \text{rref}(A) \\
[R,jb] &= \text{rref}(A,\text{tol})
\end{align*}
\]

Description
\( R = \text{rref}(A) \) produces the reduced row echelon form of \( A \) using Gauss Jordan elimination with partial pivoting. A default tolerance of \( (\max(\text{size}(A)) \cdot \text{eps} \cdot \text{norm}(A,\infty)) \) tests for negligible column elements.

\([R,jb] = \text{rref}(A)\) also returns a vector \( jb \) such that:

- \( r = \text{length}(jb) \) is this algorithm’s idea of the rank of \( A \).
- \( x(jb) \) are the pivot variables in a linear system \( Ax = b \).
- \( A(:,jb) \) is a basis for the range of \( A \).
- \( R(1:r,jb) \) is the \( r \)-by-\( r \) identity matrix.

\([R,jb] = \text{rref}(A,\text{tol})\) uses the given tolerance in the rank tests.

Roundoff errors may cause this algorithm to compute a different value for the rank than \( \text{rank}, \text{orth} \) and \( \text{null} \).

Examples
Use \text{rref} on a rank-deficient magic square:

\[
A = \text{magic}(4), \quad R = \text{rref}(A)
\]

\[
A =
\begin{bmatrix}
16 & 2 & 3 & 13 \\
5 & 11 & 10 & 8 \\
9 & 7 & 6 & 12 \\
4 & 14 & 15 & 1
\end{bmatrix}
\]

\[
R =
\begin{bmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 3
\end{bmatrix}
\]
rref

\[
\begin{bmatrix}
0 & 0 & 1 & -3 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

See Also

inv, lu, rank
Purpose
Convert real Schur form to complex Schur form

Syntax
[U,T] = rsf2csf(U,T)

Description
The complex Schur form of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The real Schur form has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

[U,T] = rsf2csf(U,T) converts the real Schur form to the complex form.

Arguments U and T represent the unitary and Schur forms of a matrix A, respectively, that satisfy the relationships: \( A = U*T*U' \) and \( U'*U = \text{eye}(\text{size}(A)) \). See schur for details.

Examples
Given matrix A,

\[
\begin{bmatrix}
1 & 1 & 1 & 3 \\
1 & 2 & 1 & 1 \\
1 & 1 & 3 & 1 \\
-2 & 1 & 1 & 4
\end{bmatrix}
\]

with the eigenvalues

\[
4.8121 \quad 1.9202 + 1.4742i \quad 1.9202 + 1.4742i \quad 1.3474
\]

Generating the Schur form of A and converting to the complex Schur form

\[
[u,t] = \text{schur}(A);
[U,T] = \text{rsf2csf}(u,t)
\]

yields a triangular matrix T whose diagonal (underlined here for readability) consists of the eigenvalues of A.

\[
\begin{bmatrix}
2 & -1 & 1 & 1 \\
0 & 2 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
2 & -3 & 1 & 1 \\
0 & 2 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
rsf2csf

\[
-0.4916 - 0.2756 - 0.4411i \\
-0.4980 - 0.1012 + 0.2163i \\
-0.6751 0.1842 + 0.3860i \\
-0.2337 0.2635 - 0.6481i
\]

\[
0.2133 + 0.5699i \\
-0.1046 + 0.2093i \\
-0.1867 - 0.3808i \\
0.3134 - 0.5448i
\]

\[
-0.4411 - 0.2756 + 0.4916i \\
-0.2133 + 0.5699i \\
-0.1046 + 0.2093i \\
0.3134 - 0.5448i
\]

\[
-0.3428 \\
0.8001 \\
-0.4260 \\
0.2466
\]

\[T = \]

\[
4.8121 - 0.9697 + 1.0778i \\
0 1.9202 + 1.4742i \\
0 0 1.9202 - 1.4742i \\
0 0 0 1.3474
\]

\[
-0.5212 + 2.0051i \\
2.3355 \\
1.9202 - 1.4742i \\
0
\]

\[
-1.0067 \\
0.1117 + 1.6547i \\
0.8002 + 0.2310i \\
1.3474
\]

**See Also**

schur
**Purpose**
Run script that is not on current path

**Syntax**
run scriptname

**Description**
run scriptname runs the MATLAB script specified by scriptname. If scriptname contains the full pathname to the script file, then run changes the current directory to be the one in which the script file resides, executes the script, and sets the current directory back to what it was. The script is run within the caller’s workspace.

run is a convenience function that runs scripts that are not currently on the path. Typically, you just type the name of a script at the MATLAB prompt to execute it. This works when the script is on your path. Use the cd or addpath function to make a script executable by entering the script name alone.

**See Also**
cd, addpath
**Purpose**

Save workspace variables to disk

**Graphical Interface**

As an alternative to the `save` function, select **Save Workspace As** from the **File** menu in the MATLAB desktop, or use the Workspace browser.

**Syntax**

```
save
save filename
save filename content
save filename options
save filename content options
save('filename', 'var1', 'var2', ...)
```

**Description**

`save` stores all variables from the current MATLAB workspace in a MATLAB formatted file (MAT-file) named `matlab.mat` that resides in the current working directory. Use the `load` function to retrieve data stored in MAT-files. By default, MAT-files are double-precision, binary files. You can create a MAT-file on one machine and then load it on another machine using a different floating-point format, and retaining as much accuracy and range as the different formats allow. MAT-files can also be manipulated by other programs external to MATLAB.

`save filename` stores all variables in the current workspace in the file `filename`. If you do not specify an extension to the filename, MATLAB uses `.mat`. The file must be writable. To save to another directory, use a full pathname for the `filename`.

`save filename content` stores only those variables specified by `content` in file `filename`. If `filename` is not specified, MATLAB stores the data in a file called `matlab.mat`. See the following table.

<table>
<thead>
<tr>
<th><strong>Values for content</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>varlist</code></td>
<td>Save only those variables that are in <code>varlist</code>. You can use the * wildcard to save only those variables that match the specified pattern. For example, <code>save('A*')</code> saves all variables that start with A.</td>
</tr>
</tbody>
</table>
In this table, the terms varlist, exprlist, and fieldlist refer to one or more variable names, regular expressions, or structure field names separated by either spaces or commas, depending on whether you are using the MATLAB command or function format. See the examples below:

Command format:

    save myfile.mat firstname lastname

Function format:

    save('myfile.mat', 'firstname', 'lastname')

save filename options stores all variables from the MATLAB workspace in file filename according to one or more of the following options. If filename is not specified, MATLAB stores the data in a file called matlab.mat.

---

### Values for **options**

<table>
<thead>
<tr>
<th>Description</th>
<th>Values for <strong>options</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>-append</td>
<td>Add new variables to those already stored in an existing MAT-file.</td>
</tr>
</tbody>
</table>
Values for options | Description
--- | ---
-format | Save using the specified binary or ASCII format. See the section on, “MAT-File Format Options” on page 2-3178, below.
-version | Save in a format that can be loaded into an earlier version of MATLAB. See the section on “Version Compatibility Options” on page 2-3178, below.

save filename content options stores only those variables specified by content in file filename, also applying the specified options. If filename is not specified, MATLAB stores the data in a file called matlab.mat.

save('filename', 'var1', 'var2', ...) is the function form of the syntax.

**MAT-File Format Options**

The following table lists the valid MAT-file format options.

<table>
<thead>
<tr>
<th>MAT-file format Options</th>
<th>How Data Is Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ascii</td>
<td>Save data in 8-digit ASCII format.</td>
</tr>
<tr>
<td>-ascii -tabs</td>
<td>Save data in 8-digit ASCII format delimited with tabs.</td>
</tr>
<tr>
<td>-ascii -double</td>
<td>Save data in 16-digit ASCII format.</td>
</tr>
<tr>
<td>-ascii -double -tabs</td>
<td>Save data in 16-digit ASCII format delimited with tabs.</td>
</tr>
<tr>
<td>-mat</td>
<td>Binary MAT-file form (default).</td>
</tr>
</tbody>
</table>

**Version Compatibility Options**

The following table lists version compatibility options. These options enable you to save your workspace data to a MAT-file that can then be loaded into an earlier version of MATLAB software. The resulting
MAT-file supports only those data items and features that were available in this earlier version of MATLAB. (See the second table below for what is supported in each version.)

<table>
<thead>
<tr>
<th><strong>version</strong> Option</th>
<th><strong>Use When Running ...</strong></th>
<th><strong>To Save a MAT-File That You Can Load In ...</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>-v7.3</td>
<td>Version 7.3 or later</td>
<td>Version 7.3 or later</td>
</tr>
<tr>
<td>-v7</td>
<td>Version 7.3 or later</td>
<td>Versions 7.0 through 7.2 (or later)</td>
</tr>
<tr>
<td>-v6</td>
<td>Version 7 or later</td>
<td>Versions 5 and 6 (or later)</td>
</tr>
<tr>
<td>-v4</td>
<td>Version 5 or later</td>
<td>Versions 1 through 4 (or later)</td>
</tr>
</tbody>
</table>

The default version option is the value specified in the **Preferences** dialog box. Select **File > Preferences** in the Command Window, click **General**, and then **MAT-Files** to view or change the default.

The next table shows what data items and features are supported in different versions of MATLAB. You can use this information to determine which of the version compatibility options shown above to use.

<table>
<thead>
<tr>
<th><strong>MATLAB Versions</strong></th>
<th><strong>Data Items or Features Supported</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>4 and earlier</td>
<td>Support for 2D double, character, and sparse</td>
</tr>
<tr>
<td>5 and 6</td>
<td>Version 4 capability plus support for ND arrays, structs, and cells</td>
</tr>
<tr>
<td>7.0 through 7.2</td>
<td>Version 6 capability plus support for data compression and Unicode character encoding</td>
</tr>
<tr>
<td>7.3 and later</td>
<td>Version 7.2 capability plus support for data items greater than or equal to 2GB</td>
</tr>
</tbody>
</table>
Remarks

When using the -regexp switch, save considers all variables in the argument list, with the exception of the optional filename and structure name variables, to be regular expressions. The filename, if specified, is always the first argument in the argument list, provided that this argument is a variable name. The structure name, if specified, is always the first argument following the -struct keyword, provided that the argument list includes that keyword.

When working on 64-bit platforms, you can have data items in your workspace that occupy more than 2 GB. To save data of this size, you must use the HDF5-based version of the MATLAB MAT-file. Use the v7.3 option to do this:

```
save -v7.3 myfile v1 v2
```

If you are running MATLAB on a 64-bit computer system and you attempt to save a variable that is too large for a version 7 (or earlier) MAT-file, that is, you save without using the -v7.3 option, MATLAB skips that variable during the save operation and issues a warning message to that effect.

If you are running MATLAB on a 32-bit computer system and attempt to load a variable from a -v7.3 MAT-file that is too large to fit in 32-bit address space, MATLAB skips that variable and issues a warning message to that effect.

MAT-files saved with compression and Unicode encoding cannot be loaded into versions of MATLAB prior to MATLAB Version 7.0. If you save data to a MAT-file that you intend to load using MATLAB Version 6 or earlier, you must specify the -v6 option when saving. This disables compression and Unicode encoding for that particular save operation.

If you want to save to a file that you can then load into a Version 4 MATLAB session, you must use the -v4 option when saving. When you use this option, variables that are incompatible with MATLAB Version 4 are not saved to the MAT-file. For example, ND arrays, structs, cells, etc. cannot be saved to a MATLAB Version 4 MAT-file. Also, variables with names that are longer than 19 characters cannot be saved to a MATLAB Version 4 MAT-file.
For information on any of the following topics related to saving to MAT-files, see “Exporting Data to MAT-Files” in the MATLAB Programming Fundamentals documentation:

-Appending variables to an existing MAT-file
-Compressing data in the MAT-file
-Saving in ASCII format
-Saving in MATLAB Version 4 format
-Saving with Unicode character encoding
-Data storage requirements
-Saving from external programs

For information on saving figures, see the documentation for `hgsave` and `saveas`. For information on exporting figures to other graphics formats, see the documentation for `print`.

**Examples**

**Example 1**

Save all variables from the workspace in binary MAT-file `test.mat`:

```matlab
save test.mat
```

**Example 2**

Save variables `p` and `q` in binary MAT-file `test.mat`.

In this example, the file name is stored in a variable, `savefile`. You must call `save` using the function syntax of the command if you intend to reference the file name through a variable.

```matlab
savefile = 'test.mat';
p = rand(1, 10);
q = ones(10);
save(savefile, 'p', 'q')
```
**Example 3**

Save the values of variables `vol` and `temp` in ASCII format to a file named `june10`:

```matlab
save('d:\mymfiles\june10','vol','temp','-ASCII')
```

**Example 4**

Save the fields of structure `s1` as individual variables rather than as an entire structure.

```matlab
s1.a = 12.7; s1.b = {'abc', [4 5; 6 7]}; s1.c = 'Hello!';
save newstruct.mat -struct s1;
clear
```

Check what was saved to `newstruct.mat`:

```matlab
whos -file newstruct.mat
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
<tr>
<td>b</td>
<td>1x2</td>
<td>158</td>
<td>cell array</td>
</tr>
<tr>
<td>c</td>
<td>1x6</td>
<td>12</td>
<td>char array</td>
</tr>
</tbody>
</table>

Grand total is 16 elements using 178 bytes

Read only the `b` field into the MATLAB workspace.

```matlab
str = load('newstruct.mat', 'b')
str =
    b: {'abc' [2x2 double]}
```

**Example 5**

Using regular expressions, save in MAT-file `mydata.mat` those variables with names that begin with `Mon`, `Tue`, or `Wed`:

```matlab
save('mydata', '-regexp', '^Mon|^Tue|^Wed');
```
Here is another way of doing the same thing. In this case, there are three separate expression arguments:

```matlab
save('mydata', '-regexp', '^Mon', '^Tue', '^Wed');
```

**Example 6**

Save a 3000-by-3000 matrix uncompressed to file `c1.mat`, and compressed to file `c2.mat`. The compressed file uses about one quarter the disk space required to store the uncompressed data:

```matlab
x = ones(3000);
y = uint32(rand(3000) * 100);

save -v6 c1 x y % Save without compression
save -v7 c2 x y % Save with compression

d1 = dir('c1.mat');
d2 = dir('c2.mat');

d1.bytes
ans =
    45000240 % Size of the uncompressed data in bytes.
d2.bytes
ans =
    11985283 % Size of the compressed data in bytes.

d2.bytes/d1.bytes
ans =
    0.2663 % Ratio of compressed to uncompressed
```

**See Also**

`load`, `clear`, `diary`, `fileformats`, `fprintf`, `fwrite`, `genvarname`, `who`, `whos`, `workspace`, `regexp`
save (COM)

**Purpose**
Serialize control object to file

**Syntax**

```matlab
h.save('filename')
save(h, 'filename')
```

**Description**

`h.save('filename')` saves the COM control object, `h`, to the file specified in the string, `filename`.

`save(h, 'filename')` is an alternate syntax for the same operation.

**Note**
The COM save function is only supported for controls at this time.

**Remarks**
COM functions are available on Microsoft Windows systems only.

**Examples**

Create an mwsamp control and save its original state to the file mwsample:

```matlab
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.save('mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```matlab
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```matlab
h.load('mwsample');
h.get
```

MATLAB displays the original values:

```matlab
ans =
   Label: 'Label'
  Radius: 20
```
See Also

load (COM), actxcontrol, actxserver, release, delete (COM)
**Purpose**
Save serial port objects and variables to MAT-file

**Syntax**
save filename
save filename obj1 obj2...

**Description**
save filename saves all MATLAB variables to the MAT-file filename. If an extension is not specified for filename, then the .mat extension is used.

save filename obj1 obj2... saves the serial port objects obj1 obj2... to the MAT-file filename.

**Remarks**
You can use save in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and serial port objects as strings. For example, to save the serial port object s to the file MySerial.mat on a Windows platform,

```matlab
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the serial port object is not automatically stored in the MAT-file. For example, suppose there is data in the input buffer for obj. To save that data to a MAT-file, you must bring it into the MATLAB workspace using one of the synchronous read functions, and then save to the MAT-file using a separate variable name. You can also save data to a text file with the record function.

You return objects and variables to the MATLAB workspace with the load command. Values for read-only properties are restored to their default values upon loading. For example, the Status property is restored to closed. To determine if a property is read-only, examine its reference pages.

**Example**
This example illustrates how to use the command and functional form of save on a Windows platform.

```matlab
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
```
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')

See Also

Functions
load, record

Properties
Status
Purpose
Save figure or Simulink block diagram using specified format

GUI Alternative
Use File > Save As on the figure window menu to access the Save As dialog, in which you can select a graphics format. For details, see “Exporting in a Specific Graphics Format” in the MATLAB Graphics documentation. Sizes of files written to image formats by this GUI and by saveas can differ due to disparate resolution settings.

Syntax
```
saveas(h,'filename.ext')
saveas(h,'filename','format')
```

Description
saveas(h,'filename.ext') saves the figure or Simulink block diagram with the handle h to the file filename.ext. The format of the file is determined by the extension, ext. Allowable values for ext are listed in this table.

You can pass the handle of any Handle Graphics object to saveas, which then saves the parent figure to the object you specified should h not be a figure handle. This means that saveas cannot save a subplot without also saving all subplots in its parent figure.

<table>
<thead>
<tr>
<th>ext Value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>ai</td>
<td>Adobe® Illustrator ‘88</td>
</tr>
<tr>
<td>bmp</td>
<td>Windows bitmap</td>
</tr>
<tr>
<td>emf</td>
<td>Enhanced metafile</td>
</tr>
<tr>
<td>eps</td>
<td>EPS Level 1</td>
</tr>
<tr>
<td>fig</td>
<td>MATLAB figure (invalid for Simulink block diagrams)</td>
</tr>
<tr>
<td>jpg</td>
<td>JPEG image (invalid for Simulink block diagrams)</td>
</tr>
<tr>
<td>m</td>
<td>MATLAB M-file (invalid for Simulink block diagrams)</td>
</tr>
<tr>
<td>pbm</td>
<td>Portable bitmap</td>
</tr>
</tbody>
</table>
saveas(h,'filename','format') saves the figure or Simulink block diagram with the handle h to the file called filename using the specified format. The filename can have an extension, but the extension is not used to define the file format. If no extension is specified, the standard extension corresponding to the specified format is automatically appended to the filename.

Allowable values for format are the extensions in the table above and the device drivers and graphic formats supported by print. The drivers and graphic formats supported by print include additional file formats not listed in the table above. When using a print device type to specify format for saveas, do not prefix it with -d.

**Remarks**

You can use open to open files saved using saveas with an m or fig extension. Other saveas and print formats are not supported by open. Both the **Save As** and **Export Setup** dialog boxes that you access from a figure's **File** menu use saveas with the **format** argument, and support all device and file types listed above.
**Note** Whenever you specify a format for saving a figure with the *Save As* menu item, that file format is used again the next time you save that figure or a new one. If you do not want to save in the previously-used format, use *Save As* and be sure to set the *Save as type* drop-down menu to the kind of file you want to write. However, saving a figure with the `saveas` function and a format does not change the *Save as type* setting in the GUI.

If you want to control the size or resolution of figures saved in image (bit-mapped) formats, such as BMP or JPG, use the `print` command and specify dots-per-inch resolution with the `r` switch.

**Examples**

**Example 1: Specify File Extension**

Save the current figure that you annotated using the Plot Editor to a file named `pred_prey` using the MATLAB `fig` format. This allows you to open the file `pred_prey.fig` at a later time and continue editing it with the Plot Editor.

```matlab
saveas(gcf,'pred_prey.fig')
```

**Example 2: Specify File Format but No Extension**

Save the current figure, using Adobe Illustrator format, to the file `logo`. Use the `ai` extension from the above table to specify the format. The file created is `logo.ai`.

```matlab
saveas(gcf,'logo', 'ai')
```

This is the same as using the Adobe Illustrator format from the print devices table, which is `-dill`; use `doc print` or `help print` to see the table for print device types. The file created is `logo.ai`. MATLAB automatically appends the `ai` extension for an Illustrator format file because no extension was specified.

```matlab
saveas(gcf,'logo', 'ill')
```
Example 3: Specify File Format and Extension

Save the current figure to the file \texttt{star.eps} using the Level 2 Color PostScript format. If you use \texttt{doc print} or \texttt{help print}, you can see from the table for print device types that the device type for this format is \texttt{-dpsc2}. The file created is \texttt{star.eps}.

\begin{verbatim}
saveas(gcf,'star.eps', 'psc2')
\end{verbatim}

In another example, save the current Simulink block diagram to the file \texttt{trans.tiff} using the TIFF format with no compression. From the table for print device types, you can see that the device type for this format is \texttt{-dtiffn}. The file created is \texttt{trans.tiff}.

\begin{verbatim}
saveas(gcf,'trans.tiff', 'tiffn')
\end{verbatim}

See Also

\texttt{hgsave, open, print}

“Printing” on page 1-99 for related functions

Simulink users, see also \texttt{save_system}
**Purpose**
Modify how `save` function saves objects

**Syntax**

```matlab
B = saveobj(A)
```

**Description**

`B = saveobj(A)` is called by the MATLAB `save` function when object `A` is saved to a MAT-file. This call executes the object’s `saveobj` method, if such a method exists. The returned value `B` is written to the MAT-file by `save`. If you define a `saveobj` method for your class, you should define a `loadobj` method to take the appropriate action with the class is loaded.

Define a `saveobj` method to modify the object before the save operation. For example, you might define a `saveobj` method that saves all property data in a struct. You `loadobj` method then would reconstruct the object from the data in the struct.

**Remarks**

MATLAB does not call the superclass `saveobj` methods of an object when you save it. However, you should call the superclass `saveobj` method from the subclass implementation of `saveobj`. For example, the `mySub` class calls its superclass `saveobj` method with the following syntax:

```matlab
classdef mySub < super
    methods
        function sobj = saveobj(obj)
            % Call superclass saveobj method
            sobj = saveobj@super(obj);
            % Perform subclass save operations
            ...
        end
        ...
        end
    end
    ...
end
```

Subclasses should always define a `saveobj` method if their superclass defines a `saveobj` method. See “The Save and Load Process” for more information.
MATLAB invokes `saveobj` separately for each object saved.

**Examples**

The following example shows a `saveobj` method that determines if an object has already been assigned an account number from a previous `save` operation. If not, `saveobj` calls the object's `getAccountNumber` method to obtain an account number and assigns it to the `AccountNumber` property. The value returned by `saveobj` (b) is saved to the MAT-file.

```matlab
function b = saveobj(a)
    if isempty(a.AccountNumber)
        a.AccountNumber = getAccountNumber(a);
    end
    b = a;
end
```

**See Also**

`save`, `load`, `loadobj`
**Purpose**
Save current search path

**GUI Alternatives**
As an alternative to the `savepath` function, use the Set Path dialog box.

**Syntax**
```
savepath
directory/pathdef.m
```

**Description**
`savepath` saves the current search path for the MATLAB program so you can use it in a future session. MATLAB saves the search path to the `pathdef.m` file from which it created the search path at startup. If there is a `pathdef.m` file located in the current directory, `savepath` saves the current search path to it instead of the search path it created at startup. Consider using `savepath` in your `finish.m` file to automatically save the search path when you exit MATLAB.

`savepath directory/pathdef.m` saves the current search path to `pathdef.m` located in `directory`. Use this form of the syntax if you want an alternative to the standard path for use in a future session. If you do not specify `directory`, MATLAB saves `pathdef.m` in the current directory. `directory` can be a relative or absolute path. If you want to use the saved path in a future session, make `directory` be the startup directory for MATLAB.

**Examples**
The statement
```
savepath I:\my_matlab_files\pathdef.m
```
saves the current search path to `pathdef.m`, located in `I:\my_matlab_files`.

**See Also**
`addpath`, `cd`, `dir`, `finish`, `genpath`, `matlabroot`, `partialpath`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `startup`, `userpath`, `what`

Search Path and “Startup Directory for the MATLAB Program” in the MATLAB Desktop Tools and Development Environment documentation
Purpose

Scatter plot

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

scatter(X,Y,S,C)
scatter(X,Y)
scatter(X,Y,S)
scatter(...,markertype)
scatter(...,'filled')
scatter(...,'PropertyName',propertyvalue)
h = scatter(...)
hpatch = scatter('v6',...)

Description

scatter(X,Y,S,C) displays colored circles at the locations specified by the vectors X and Y (which must be the same size).

S determines the area of each marker (specified in points^2). S can be a vector the same length as X and Y or a scalar. If S is a scalar, MATLAB draws all the markers the same size. If S is empty, the default size is used.

C determines the color of each marker. When C is a vector the same length as X and Y, the values in C are linearly mapped to the colors in the current colormap. When C is a 1-by-3 matrix, it specifies the colors of the markers as RGB values. If you have 3 points in the scatter plot and wish to have the colors be indices into the colormap, C should be a 3-by-1 matrix. C can also be a color string (see ColorSpec for a list of color string specifiers).
**scatter**

scatter(X,Y) draws the markers in the default size and color.

scatter(X,Y,S) draws the markers at the specified sizes (S) with a single color. This type of graph is also known as a bubble plot.

scatter(...,markertype) uses the marker type specified instead of 'o' (see LineSpec for a list of marker specifiers).

scatter(...,'filled') fills the markers.

scatter(...,'PropertyName',propertyvalue) creates the scatter graph, applying the specified property settings. See scattergroup properties for a description of properties.

scatter(axes_handles,...) plots into the axes object with handle axes_handle instead of the current axes object (gca).

h = scatter(...) returns the handle of the scattergroup object created.

**Backward-Compatible Version**

hpatch = scatter('v6',...) returns the handles to the patch objects created by scatter (see Patch Properties for a list of properties you can specify using the object handles and set).

---

**Note** The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

---

See Plot Objects and Backward Compatibility for more information.

**Example**

```
load seamount
scatter(x,y,5,z)
```
See Also  

scatter3, plot3  

“Scatter/Bubble Plots” on page 1-98 for related functions  

See “Triangulations and Scattered Data” for related information.  

See Scattergroup Properties for property descriptions.
**Purpose**

3-D scatter plot

![3-D scatter plot](image)

**GUI Alternatives**

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

scatter3(X,Y,Z,S,C)
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(...,markertype)
scatter3(...,'filled')
scatter3(...,'PropertyName',propertyvalue)
h = scatter3(...)
hpatch = scatter3('v6',...)

**Description**

`scatter3(X,Y,Z,S,C)` displays colored circles at the locations specified by the vectors `X`, `Y`, and `Z` (which must all be the same size).

`S` determines the size of each marker (specified in points). `S` can be a vector the same length as `X`, `Y`, and `Z` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size.

`C` determines the color of each marker. When `C` is a vector the same length as `X` and `Y`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a 1-by-3 matrix, it specifies the colors of the markers as RGB values. If you have 3 points in the scatter plot and wish to have the colors be indices into the colormap, `C` should be a 3-by-1 matrix. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers).

`scatter3(X,Y,Z)` draws the markers in the default size and color.
\texttt{scatter3(X,Y,Z,S)} draws markers at the specified sizes (S) in a single color.

\texttt{scatter3(...,markertype)} uses the marker type specified instead of 'o' (see \texttt{LineSpec} for a list of marker specifiers).

\texttt{scatter3(...,'filled')} fills the markers.

\texttt{scatter3(...,'PropertyName',propertyvalue)} creates the scatter graph, applying the specified property settings. See \texttt{scattergroup} properties for a description of properties.

\texttt{h = scatter3(...)} returns handles to the scattergroup objects created by \texttt{scatter3}. See \texttt{Scattergroup Properties} for property descriptions.

\textbf{Backward-Compatible Version}

\texttt{hpatch = scatter3('v6',...)} returns the handles to the patch objects created by \texttt{scatter3} (see \texttt{Patch} for a list of properties you can specify using the object handles and \texttt{set}).

\textbf{Note} The \texttt{v6} option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See \texttt{Plot Objects} and Backward Compatibility for more information.

\textbf{Remarks} Use \texttt{plot3} for single color, single marker size 3-D scatter plots.

\textbf{Examples}

\begin{verbatim}
[x,y,z] = sphere(16);
X = [x(:)*.5 x(:)*.75 x(:)];
Y = [y(:)*.5 y(:)*.75 y(:)];
Z = [z(:)*.5 z(:)*.75 z(:)];
S = repmat([1 .75 .5]*10,prod(size(x)),1);
C = repmat([1 2 3],prod(size(x)),1);
scatter3(X(:),Y(:),Z(:),S(:),C(:),'filled'), view(-60,60)
\end{verbatim}
scatter3

See Also

scatter, plot3
See Scattergroup Properties for property descriptions
“Scatter/Bubble Plots” on page 1-98 for related functions
Scattergroup Properties

**Purpose**
Define scattergroup properties

**Modifying Properties**
You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default property values for scattergroup objects.

See Plot Objects for information on scattergroup objects.

**Scattergroup Property Descriptions**
This section provides a description of properties. Curly braces `{}` enclose default values.

Annotation
hg.Annotation object Read Only

*Control the display of scattergroup objects in legends.* The Annotation property enables you to specify whether this scattergroup object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the scattergroup object is displayed in a figure legend:

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Include the scattergroup object in a legend as one entry, but not its children objects</td>
</tr>
</tbody>
</table>
Scattergroup Properties

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>Do not include the scattergroup or its children in a legend (default)</td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the scattergroup as separate entries in the legend</td>
</tr>
</tbody>
</table>

Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

BeingDeleted

on | {off} Read Only

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object’s BeingDeleted property before acting.
BusyAction

cancel | {queue}

*Callback routine interruption.* The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- **cancel** — Discard the event that attempted to execute a second callback routine.
- **queue** — Queue the event that attempted to execute a second callback routine until the current callback finishes.

ButtonDownFcn

string or function handle

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.

See the figure’s SelectionType property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
• A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

CData

vector, m-by-3 matrix, ColorSpec

*Color of markers.* When CData is a vector the same length as XData and YData, the values in CData are linearly mapped to the colors in the current colormap. When CData is a length(XData)-by-3 matrix, it specifies the colors of the markers as RGB values.

CDataSource

string (MATLAB variable)

*Link YData to MATLAB variable.* Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the CData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change CData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.
Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Children
array of graphics object handles

*Children of this object.* The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object’s HandleVisibility property is set to callback or off, its handle does not show up in this object’s Children property unless you set the root ShowHiddenHandles property to on:

```
set(0,'ShowHiddenHandles','on')
```

Clipping
{on} | off

*Clipping mode.* MATLAB clips graphs to the axes plot box by default. If you set Clipping to off, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set hold to on, freeze axis scaling (axis manual), and then create a larger plot object.

CreateFcn
string or function handle

*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```
where @\texttt{CallbackFcn} is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose \texttt{CreateFcn} is being executed is accessible only through the root \texttt{CallbackObject} property, which you can query using \texttt{g cbo}.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

\texttt{DeleteFcn}

string or function handle

\textit{Callback executed during object deletion}. A callback that executes when this object is deleted (e.g., this might happen when you issue a \texttt{delete} command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose \texttt{DeleteFcn} is being executed is accessible only through the root \texttt{CallbackObject} property, which can be queried using \texttt{g cbo}.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the \texttt{BeingDeleted} property for related information.

\texttt{DisplayName}

string (default is empty string)
String used by legend for this scattergroup object. The legend function uses the string defined by the DisplayName property to label this scattergroup object in the legend.

- If you specify string arguments with the legend function, DisplayName is set to this scattergroup object’s corresponding string and that string is used for the legend.
- If DisplayName is empty, legend creates a string of the form, \[\text{data } n\], where \(n\) is the number assigned to the object based on its location in the list of legend entries. However, legend does not set DisplayName to this string.
- If you edit the string directly in an existing legend, DisplayName is set to the edited string.
- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.
- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See “Controlling Legends” for more examples.

EraseMode

\{normal\} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- normal — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
Scattergroup Properties

- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.

- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is set to `none`). That is, it isn’t erased correctly if there are objects behind it.

- **background** — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is set to `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

**Printing with Nonnormal Erase Modes**

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

You can use the MATLAB `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.
Scattergroup Properties

HandleVisibility
{on} | callback | off

*Control access to object’s handle by command-line users and GUIs.* This property determines when an object’s handle is visible in its parent’s list of children. HandleVisibility is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- on — Handles are always visible when HandleVisibility is on.
- callback — Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- off — Setting HandleVisibility to off makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

**Functions Affected by Handle Visibility**

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

**Properties Affected by Handle Visibility**

When a handle’s visibility is restricted using callback or off, the object’s handle does not appear in its parent’s `Children` property, figures do not appear in the root’s `CurrentFigure` property, objects do not appear in the root’s `CallbackObject` property or in
the figure’s `CurrentObject` property, and axes do not appear in their parent’s `CurrentAxes` property.

**Overriding Handle Visibility**

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

**Handle Validity**

Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties and pass it to any function that operates on handles.

---

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

- **HitTest**
  - `{on} | off`

  *Selectable by mouse click.* `HitTest` determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

- **HitTestArea**
  - `on | {off}`
Select the object by clicking lines or area of extent. This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When HitTestArea is off, you must click the object’s lines or markers (excluding the baseline, if any) to select the object. When HitTestArea is on, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

**Interruptible**

{on} | off

Callback routine interruption mode. The Interruptible property controls whether an object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

Setting Interruptible to on allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the gca or gcf command) when an interruption occurs.

**LineWidth**

scalar

The width of linear objects and edges of filled areas. Specify this value in points (1 point = \( \frac{1}{72} \) inch). The default LineWidth is 0.5 points.
Scattergroup Properties

**Marker**

character (see table)

*Marker symbol.* The Marker property specifies the type of markers that are displayed at plot vertices. You can set values for the Marker property independently from the LineStyle property. Supported markers include those shown in the following table.

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>o</td>
<td>Circle</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

**MarkerEdgeColor**

ColorSpec | none | {auto}

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none
specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the CData property.

MarkerFaceColor
ColorSpec | {none} | auto

*Marker face color.* The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or to the figure color if the axes Color property is set to none (which is the factory default for axes objects).

Parent
handle of parent axes, hgroup, or hgtransform

*Parent of this object.* This property contains the handle of the object’s parent. The parent is normally the axes, hgroup, or hgtransform object that contains the object.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

Selected
on | {off}

*Is object selected?* When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

SelectionHighlight
{on} | off
**Scattergroup Properties**

*Objects are highlighted when selected.* When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

**SizeData**

- **square points**

*Size of markers in square points.* This property specifies the area of the marker in the scatter graph in units of points. Since there are 72 points to one inch, to specify a marker that has an area of one square inch you would use a value of $72^2$.

**Tag**

- **string**

*User-specified object label.* The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

For example, you might create an areaseries object and set the `Tag` property.

```matlab
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, you can use `findobj` to find the object’s handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```matlab
set(findobj('Tag','area1'),'FaceColor','red')
```

**Type**

- **string (read only)**
**Scattergroup Properties**

*Type of graphics object.* This property contains a string that identifies the class of the graphics object. For stemseries objects, Type is 'hggroup'. The following statement finds all the hggroup objects in the current axes.

```matlab
t = findobj(gca,'Type','hggroup');
```

**UIContextMenu**
handle of a uicontextmenu object

*Associate a context menu with this object.* Assign this property the handle of a uicontextmenu object created in the object’s parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

**UserData**
array

*User-specified data.* This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

**Visible**
{on} | off

*Visibility of this object and its children.* By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s `Visible` property is set to off. Setting an object’s `Visible` property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.

**XData**
array

*X-coordinates of scatter markers.* The scatter function draws individual markers at each x-axis location in the XData array.
The input argument \( x \) in the \texttt{scatter} function calling syntax assigns values to \texttt{XData}.

\texttt{XDataSource}
string (MATLAB variable)

\textit{Link XData to MATLAB variable.} Set this property to a MATLAB variable that is evaluated in the base workspace to generate the \texttt{XData}.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change \texttt{XData}.

You can use the \texttt{refreshdata} function to force an update of the object’s data. \texttt{refreshdata} also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call \texttt{refreshdata}.

See the \texttt{refreshdata} reference page for more information.

\textbf{Note} If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

\texttt{YData}
scalar, vector, or matrix

\textit{Y-coordinates of scatter markers.} The scatter function draws individual markers at each \( y \)-axis location in the \texttt{YData} array.

The input argument \( y \) in the \texttt{scatter} function calling syntax assigns values to \texttt{YData}.
Scattergroup Properties

**YDataSource**

string (MATLAB variable)

*Link YData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change YData.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

---

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

**ZData**

vector of coordinates

*Z-coordinates.* A vector defining the z-coordinates for the graph. XData and YData must be the same length and have the same number of rows.

**ZDataSource**

string (MATLAB variable)
**Scattergroup Properties**

*Link ZData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the ZData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change ZData.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

---

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Purpose
Schur decomposition

Syntax
T = schur(A)
T = schur(A,flag)
[U,T] = schur(A,...)

Description
The schur command computes the Schur form of a matrix.

T = schur(A) returns the Schur matrix T.
T = schur(A,flag) for real matrix A, returns a Schur matrix T in one of two forms depending on the value of flag:

'complex'
T is triangular and is complex if A has complex eigenvalues.

'real'
T has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. 'real' is the default.

If A is complex, schur returns the complex Schur form in matrix T. The complex Schur form is upper triangular with the eigenvalues of A on the diagonal.

The function rsf2csf converts the real Schur form to the complex Schur form.

[U,T] = schur(A,...) also returns a unitary matrix U so that A = U*T*U' and U'*U = eye(size(A)).

Examples
H is a 3-by-3 eigenvalue test matrix:

H = [ -149  -50  -154
     537   180   546
      -27    -9  -25   ]

Its Schur form is

schur(H)
The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

### Algorithm

#### Input of Type Double

If \( A \) has type `double`, `schur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real symmetric</td>
<td>DSYTRD, DSTEQR</td>
</tr>
<tr>
<td></td>
<td>DSYTRD, DORGTR, DSTEQR (with output U)</td>
</tr>
<tr>
<td>Real nonsymmetric</td>
<td>DGEHRD, DHSEQR</td>
</tr>
<tr>
<td></td>
<td>DGEHRD, DORGH, DHSEQR (with output U)</td>
</tr>
<tr>
<td>Complex Hermitian</td>
<td>ZHETRD, ZSTEQR</td>
</tr>
<tr>
<td></td>
<td>ZHETRD, ZUNGH, ZSTEQR (with output U)</td>
</tr>
<tr>
<td>Non-Hermitian</td>
<td>ZGEHRD, ZHSEQR</td>
</tr>
<tr>
<td></td>
<td>ZGEHRD, ZUNGH, ZHSEQR (with output U)</td>
</tr>
</tbody>
</table>

#### Input of Type Single

If \( A \) has type `single`, `schur` uses the LAPACK routines listed in the following table to compute the Schur form of a matrix:

<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real symmetric</td>
<td>SSYTRD, SSTEQR</td>
</tr>
<tr>
<td></td>
<td>SSYTRD, SORGTR, SSTEQR (with output U)</td>
</tr>
<tr>
<td>Matrix A</td>
<td>Routine</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>Real nonsymmetric</td>
<td>SGEHRD, SHSEQR</td>
</tr>
<tr>
<td></td>
<td>SGEHRD, SORGHR, SHSEQR (with output U)</td>
</tr>
<tr>
<td>Complex Hermitian</td>
<td>CHETRD, CSTEQR</td>
</tr>
<tr>
<td></td>
<td>CHETRD, CUNGTR, CSTEQR (with output U)</td>
</tr>
<tr>
<td>Non-Hermitian</td>
<td>CGEHRD, CHSEQR</td>
</tr>
<tr>
<td></td>
<td>CGEHRD, CUNGHR, CHSEQR (with output U)</td>
</tr>
</tbody>
</table>

**See Also**
eig, hess, qz, rsf2csf

**References**
Purpose
Script M-file description

Description
A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, you can obtain subsequent MATLAB input from the file. Script files have a filename extension of .m and are often called M-files.

Scripts are the simplest kind of M-file. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, so you can use them in further computations. In addition, scripts can produce graphical output using commands like plot.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any M-file, scripts can contain comments. Any text following a percent sign (%) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also
echo, function, type
### Purpose
Secant of argument in radians

### Syntax
Y = sec(X)

### Description
The sec function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

Y = sec(X) returns an array the same size as X containing the secant of the elements of X.

### Examples
Graph the secant over the domains $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$.

```matlab
x1 = -pi/2+0.01:0.01:pi/2-0.01;
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;
plot(x1,sec(x1),x2,sec(x2)), grid on
```
The expression $\text{sec}(\pi/2)$ does not evaluate as infinite but as the reciprocal of the floating-point accuracy $\text{eps}$, because $\pi$ is a floating-point approximation to the exact value of $\pi$.

**Definition**

The secant can be defined as

$$\text{sec}(z) = \frac{1}{\cos(z)}$$

**Algorithm**

$\text{sec}$ uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

**See Also**

$\text{secd}$, $\text{sech}$, $\text{asec}$, $\text{asecd}$, $\text{asech}$
Purpose  
Secant of argument in degrees

Syntax  
$Y = \text{secd}(X)$

Description  
$Y = \text{secd}(X)$ is the secant of the elements of $X$, expressed in degrees. For odd integers $n$, $\text{secd}(n*90)$ is infinite, whereas $\text{sec}(n*pi/2)$ is large but finite, reflecting the accuracy of the floating point value of $\pi$.

See Also  
sec, sech, asec, asecd, asech
**Purpose**
Hyperbolic secant

**Syntax**

\[ Y = \text{sech}(X) \]

**Description**
The `sech` function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

\[ Y = \text{sech}(X) \]
returns an array the same size as \( X \) containing the hyperbolic secant of the elements of \( X \).

**Examples**
Graph the hyperbolic secant over the domain \(-2\pi \leq x \leq 2\pi\).

```matlab
x = -2*pi:0.01:2*pi;
plot(x,sech(x)), grid on
```
Algorithm

sech uses this algorithm.

\[
\text{sech}(z) = \frac{1}{\cosh(z)}
\]

Definition

The secant can be defined as

\[
\text{sech}(z) = \frac{1}{\cosh(z)}
\]

Algorithm

sec uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

See Also

asec, asech, sec
Purpose: Select, move, resize, or copy axes and uicontrol graphics objects

Syntax:

```
A = selectmoveresize
set(gca,'ButtonDownFcn','selectmoveresize')
```

Description: `selectmoveresize` is useful as the callback routine for axes and uicontrol button down functions. When executed, it selects the object and allows you to move, resize, and copy it.

`A = selectmoveresize` returns a structure array containing:

- `A.Type`: a string containing the action type, which can be Select, Move, Resize, or Copy
- `A.Handles`: a list of the selected handles, or, for a Copy, an m-by-2 matrix containing the original handles in the first column and the new handles in the second column

```
set(gca,'ButtonDownFcn','selectmoveresize')
```
sets the `ButtonDownFcn` property of the current axes to `selectmoveresize`:

See Also:

The `ButtonDownFcn` property of axes and uicontrol objects

“Object Manipulation” on page 1-107 for related functions
**Purpose**

Semilogarithmic plots

---

**GUI Alternatives**

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

---

**Syntax**

semilogx(Y)
semilogx(X1,Y1,...)  
semilogx(X1,Y1,LineSpec,...)  
semilogx(...,'PropertyName',PropertyValue,...)

h = semilogx(...)

h = semilogy(...)

hlines = semilogx('v6',...)

---

**Description**

*semilogx* and *semilogy* plot data as logarithmic scales for the *x*- and *y*-axis, respectively.

*semilogx*(Y) creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of *Y* versus their index if *Y* contains real numbers. *semilogx*(Y) is equivalent to *semilogx*(real(Y), imag(Y)) if *Y* contains complex numbers. *semilogx* ignores the imaginary component in all other uses of this function.

*semilogy*(...) creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis.

*semilogx*(X1,Y1,...) plots all *Xn* versus *Yn* pairs. If only *Xn* or *Yn* is a matrix, *semilogx* plots the vector argument versus the rows or
semilogx, semilogy

columns of the matrix, depending on whether the vector’s row or column dimension matches the matrix.

semilogx(X1,Y1,LineSpec,...) plots all lines defined by the Xn,Yn,LineSpec triples. LineSpec determines line style, marker symbol, and color of the plotted lines.

semilogx(...,'PropertyName',PropertyValue,...) sets property values for all lineseries graphics objects created by semilogx.

h = semilogx(...) and h = semilogy(...) return a vector of handles to lineseries graphics objects, one handle per line.

**Backward-Compatible Version**

hlines = semilogx('v6',...) and hlines = semilogy('v6',...) return the handles to line objects instead of lineseries objects.

---

**Note** The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

**Remarks**

If you do not specify a color when plotting more than one line, semilogx and semilogy automatically cycle through the colors and line styles in the order specified by the current axes ColorOrder and LineStyleOrder properties.

You can mix Xn,Yn pairs with Xn,Yn,LineSpec triples; for example,

    semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)

If you attempt to add a loglog, semilogx, or semilogy plot to a linear axis mode graph with hold on, the axis mode will remain as it is and the new data will plot as linear.
Examples

Create a simple semilogy plot.

\[
x = 0:.1:10;
\]
\[
\text{semilogy}(x,10.^x)
\]

See Also

line, LineSpec, loglog, plot

“Basic Plots and Graphs” on page 1-93 for related functions
**sendmail**

**Purpose**
Send e-mail message to address list

**Syntax**

```matlab
sendmail('recipients','subject')
sendmail('recipients','subject','message','attachments')
```

**Description**

`sendmail('recipients','subject')` sends e-mail to recipients with the specified subject. For `recipients`, use a string for a single address, or a cell array of strings for multiple addresses.

`sendmail('recipients','subject','message','attachments')` sends `message` to recipients with the specified subject. For `recipients`, use a string for a single address, or a cell array of strings for multiple addresses. For `message`, use a string or cell array. When `message` is a string, the text automatically wraps at 75 characters. When `message` is a cell array, it does not wrap but rather each cell is a new line. To force text to start on a new line in strings or cells, use 10, as shown in the “Example of `sendmail` with New Lines Specified” on page 2-3233. Specify `attachments` as a cell array of files to send along with `message`.

To use `sendmail`, you must set the preferences for your e-mail server (Internet SMTP server) and your e-mail address must be set. The MATLAB software tries to read the SMTP mail server from your system registry, but if it cannot, it results in an error. In this event, identify the outgoing mail server for your electronic mail application, which is usually listed in the application’s preferences, or, consult your e-mail system administrator. Then provide the information to MATLAB using:

```matlab
setpref('Internet','SMTP_Server','myserver.myhost.com');
```

If you cannot easily determine your e-mail server, try using `mail`, as in:

```matlab
setpref('Internet','SMTP_Server','mail');
```

which might work because `mail` is often a default for mail systems.

Similarly, if MATLAB cannot determine your e-mail address and produces an error, specify your e-mail address using:
setpref('Internet','E_mail','myaddress@example.com');

**Note** The sendmail function does not support e-mail servers that require authentication.

**Examples**

**Example of sendmail with Two Attachments**

sendmail('user@otherdomain.com', ...
  'Test subject','Test message',...
  {'directory/attach1.html','attach2.doc'});

**Example of sendmail with New Lines Specified**

This mail message forces the message to start new lines after each 10.

sendmail('user@otherdomain.com','New subject', ...
            ['Line1 of message' 10 'Line2 of message' 10 ... 
            'Line3 of message' 10 'Line4 of message']);

The resulting message is:

Line1 of message
Line2 of message
Line3 of message
Line4 of message

**See Also**

getpref, setpref
### Purpose
Create serial port object

### Syntax
```
obj = serial('port')
obj = serial('port', 'PropertyName', PropertyValue,...)
```

### Description
`obj = serial('port')` creates a serial port object associated with the serial port specified by `port`. If `port` does not exist, or if it is in use, you will not be able to connect the serial port object to the device.

`Port` object name will depend upon the platform that the serial port is on. `insthwinfo ('serial')` provides a list of available serial ports. This list is an example of serial constructors on different platforms:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Serial Port Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux and Linux 64</td>
<td><code>serial('/dev/ttyS0');</code></td>
</tr>
<tr>
<td>Mac OS X and Mac OS X 64</td>
<td><code>serial('/dev/tty.KeySerial1');</code></td>
</tr>
<tr>
<td>Solaris 64</td>
<td><code>serial('/dev/term/a');</code></td>
</tr>
<tr>
<td>Windows 32 and Windows 64</td>
<td><code>serial('com1');</code></td>
</tr>
</tbody>
</table>

`obj = serial('port', 'PropertyName', PropertyValue,...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

### Remarks
When you create a serial port object, these property values are automatically configured:

- The `Type` property is given by `serial`.
- The `Name` property is given by concatenating `Serial` with the port specified in the `serial` function.
- The `Port` property is given by the port specified in the `serial` function.
You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid on a Windows platform.

```matlab
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Refer to Configuring Property Values for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to obj with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

### Example

This example creates the serial port object `s1` associated with the serial port COM1 on a Windows platform.

```matlab
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```matlab
get(s1,{'Type','Name','Port'})
ans =
    'serial'    'Serial-COM1'    'COM1'
```

To specify properties during object creation

```matlab
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

### See Also

**Functions**

`fclose`, `fopen`
serial

<table>
<thead>
<tr>
<th>Properties</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name, Port, Status</td>
<td>Type</td>
</tr>
</tbody>
</table>

2-3236
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Send break to device connected to serial port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>serialbreak(obj)</code>&lt;br&gt;<code>serialbreak(obj,time)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>serialbreak(obj)</code> sends a break of 10 milliseconds to the device connected to the serial port object, <code>obj</code>.&lt;br&gt;&lt;br&gt;<code>serialbreak(obj,time)</code> sends a break to the device with a duration, in milliseconds, specified by <code>time</code>. Note that the duration of the break might be inaccurate under some operating systems.</td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td>For some devices, the break signal provides a way to clear the hardware buffer.&lt;br&gt;&lt;br&gt;Before you can send a break to the device, it must be connected to <code>obj</code> with the <code>fopen</code> function. A connected serial port object has a <code>Status</code> property value of <code>open</code>. An error is returned if you attempt to send a break while <code>obj</code> is not connected to the device.&lt;br&gt;&lt;br&gt;<code>serialbreak</code> is a synchronous function, and blocks the command line until execution is complete.&lt;br&gt;&lt;br&gt;If you issue <code>serialbreak</code> while data is being asynchronously written, an error is returned. In this case, you must call the <code>stopasync</code> function or wait for the write operation to complete.</td>
</tr>
</tbody>
</table>

**See Also**<br><br>**Functions**<br>`fopen`, `stopasync`<br><br>**Properties**<br>`Status`
**Purpose**
Set Handle Graphics object properties

**Syntax**

```matlab
set(H,'PropertyName',PropertyValue,...)
set(H,a)
set(H,pn,pv,...)
set(H,pn,MxN_pv)
```

```matlab
a = set(h)
pv = set(h,'PropertyName')
```

**Description**

**Note** Do not use the `set` function on Java objects as it will cause a memory leak. For more information, see “Accessing Private and Public Data”

set(H,'PropertyName',PropertyValue,...) sets the named properties to the specified values on the object(s) identified by H. H can be a vector of handles, in which case `set` sets the properties’ values for all the objects.

set(H,a) sets the named properties to the specified values on the object(s) identified by H. a is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties.

set(H,pn,pv,...) sets the named properties specified in the cell array pn to the corresponding value in the cell array pv for all objects identified in H.

set(H,pn,MxN_pv) sets n property values on each of m graphics objects, where m = length(H) and n is equal to the number of property names contained in the cell array pn. This allows you to set a given group of properties to different values on each object.

a = set(h) returns the user-settable properties and possible values for the object identified by h. a is a structure array whose field names are the object’s property names and whose field values are the possible values of the corresponding properties. If you do not specify an output
argument, the MATLAB software displays the information on the screen. \( h \) must be scalar.

\[
 pv = \text{set}(h, 'PropertyName')
\]
returns the possible values for the named property. If the possible values are strings, \text{set} \ returns each in a cell of the cell array \( pv \). For other properties, \text{set} \ returns a statement indicating that \( PropertyName \) does not have a fixed set of property values. If you do not specify an output argument, MATLAB displays the information on the screen. \( h \) must be scalar.

**Remarks**

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to \text{set}.

**Setting Property Units**

Note that if you are setting both the FontSize and the FontUnits properties in one function call, you must set the FontUnits property first so that the MATLAB software can correctly interpret the specified FontSize. The same applies to figure and axes units — always set the Units property before setting properties whose values you want to be interpreted in those units. For example,

\[
 f = \text{figure}(\text{'Units','characters'},
\quad \text{'Position',[30 30 120 35]})
\]

**Examples**

Set the Color property of the current axes to blue.

\[
 \text{axes;}
\quad \text{set(gca,'Color','b')}
\]

Change all the lines in a plot to black.

\[
 \text{plot(peaks)}
\quad \text{set(findobj('Type','line'),'Color','k')}
\]

You can define a group of properties in a structure to better organize your code. For example, these statements define a structure called \text{active}, which contains a set of property definitions used for the
uicontrol objects in a particular figure. When this figure becomes the current figure, MATLAB changes the colors and enables the controls.

    active.BackgroundColor = [.7 .7 .7];
    active.Enable = 'on';
    active.ForegroundColor = [0 0 0];

    if gcf == control_fig_handle
        set(findobj(control_fig_handle,'Type','uicontrol'),active)
    end

You can use cell arrays to set properties to different values on each object. For example, these statements define a cell array to set three properties,

    PropName(1) = {'BackgroundColor'};
    PropName(2) = {'Enable'};
    PropName(3) = {'ForegroundColor'};

These statements define a cell array containing three values for each of three objects (i.e., a 3-by-3 cell array).

    PropVal(1,1) = {[.5 .5 .5]};
    PropVal(1,2) = {'off'};
    PropVal(1,3) = {[.9 .9 .9]};
    PropVal(2,1) = {[1 0 0]};
    PropVal(2,2) = {'on'};
    PropVal(2,3) = {[1 1 1]};
    PropVal(3,1) = {[.7 .7 .7]};
    PropVal(3,2) = {'on'};
    PropVal(3,3) = {[0 0 0]};

Now pass the arguments to set,

    set(H,PropName,PropVal)

where length(H) = 3 and each element is the handle to a uicontrol.
Setting Different Values for the Same Property on Multiple Objects

Suppose you want to set the value of the Tag property on five line objects, each to a different value. Note how the value cell array needs to be transposed to have the proper shape.

```matlab
h = plot(rand(5));
set(h,{'Tag'},{'line1','line2','line3','line4','line5'})
```

See Also
findobj, gca, gcf, gco, gcbo, get

“Graphics Object Identification” on page 1-100 for related functions
set (COM)

Purpose
Set object or interface property to specified value

Syntax
h.set('pname', value)
h.set('pname1', value1, 'pname2', value2, ...)
set(h, ...)

Description
h.set('pname', value) sets the property specified in the string pname to the given value.

h.set('pname1', value1, 'pname2', value2, ...) sets each property specified in the pname strings to the given value.

set(h, ...) is an alternate syntax for the same operation.

See “Handling COM Data in MATLAB Software” in the External Interfaces documentation for information on how MATLAB converts workspace matrices to COM data types.

Remarks
COM functions are available on Microsoft Windows systems only.

Examples
Create an mwsamp control and use set to change the Label and Radius properties:

f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);

h.set('Label', 'Click to fire event', 'Radius', 40);
h.invoke('Redraw');

Here is another way to do the same thing, only without set and invoke:

h.Label = 'Click to fire event';
h.Radius = 40;
h.Redraw;

See Also
get (COM), inspect, isprop, addproperty, deleteproperty
Purpose
Assign property values to handle objects derived from hgsetget class

Syntax
set(H, 'PropertyName', value, ...)
pv = set(h, 'PropertyName')
S = set(h)

Description
set(H, 'PropertyName', value, ...) sets the named property to the specified value for the objects in the handle array H.

pv = set(h, 'PropertyName') returns the possible values for the named property.

S = set(h) returns the user-settable properties and possible values for the handle object h. S is a struct whose field names are the object’s property names and whose field values are cell arrays containing the possible values of the corresponding properties having finite possible values.

See Also
See “Implementing a Set/Get Interface for Properties”
handle, hgsetget, set, get (hgsetget)
**set (RandStream)**

**Purpose**
Set random stream property

**Class**
@RandStream

**Syntax**
- `set(S,'PropertyName',Value)`
- `set(S,'Property1',Value1,'Property2',Value2,...)`
- `set(S,A)`
  - `A=set(S,'Property')`
- `set(S)`
  - `A=set(S)`
  - `set(S)`

**Description**
- `set(S,'PropertyName',Value)` sets the property 'PropertyName' of the random stream S to the value Value.
- `set(S,'Property1',Value1,'Property2',Value2,...)` sets multiple random stream property values with a single statement.
- `set(S,A)` where A is a structure whose field names are property names of the random stream S sets the properties of S named by each field with the values contained in those fields.
- `A=set(S,'Property')` or `set(S,'Property')` displays possible values for the specified property of S.
- `A=set(S)` or `set(S)` displays or returns all properties of S and their possible values.

**See Also**
@RandStream, get (RandStream), rand, randn, randi
Purpose

Configure or display serial port object properties

Syntax

set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)

Description

set(obj) displays all configurable properties values for the serial port object, obj. If a property has a finite list of possible string values, then these values are also displayed.

props = set(obj) returns all configurable properties and their possible values for obj to props. props is a structure whose field names are the property names of obj, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

set(obj,'PropertyName') displays the valid values for PropertyName if it possesses a finite list of string values.

props = set(obj,'PropertyName') returns the valid values for PropertyName to props. props is a cell array of possible string values or an empty cell array if PropertyName does not have a finite list of possible values.

set(obj,'PropertyName',PropertyValue,...) configures multiple property values with a single command.

set(obj,PN,PV) configures the properties specified in the cell array of strings PN to the corresponding values in the cell array PV. PN must be a vector. PV can be m-by-n where m is equal to the number of serial port objects in obj and n is equal to the length of PN.

set(obj,S) configures the named properties to the specified values for obj. S is a structure whose field names are serial port object properties, and whose field values are the values of the corresponding properties.
Remarks

Refer to Configuring Property Values for a list of serial port object properties that you can configure with set.

You can use any combination of property name/property value pairs, structures, and cell arrays in one call to set. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if s is a serial port object, then the following commands are all valid.

```
set(s,'BaudRate')
set(s,'baudrate')
set(s,'BAUD')
```

If you use the help command to display help for set, then you need to supply the pathname shown below.

```
help serial/set
```

Examples

This example illustrates some of the ways you can use set to configure or return property values for the serial port object s, on a Windows platform.

```
s = serial('COM1');
set(s,'BaudRate',9600,'Parity','even')
set(s,{'StopBits','RecordName'},[2,'sydney.txt'])
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```
**Purpose**

Configure or display timer object properties

**Syntax**

```matlab
set(obj)
prop_struct = set(obj)
set(obj,'PropertyName')
prop_cell=set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,S)
set(obj,PN,PV)
```

**Description**

`set(obj)` displays property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object.

`prop_struct = set(obj)` returns the property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object. The return value, `prop_struct`, is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values or empty cell arrays if the property does not have a finite set of possible string values.

`set(obj,'PropertyName')` displays the possible values for the specified property, `PropertyName`, of timer object `obj`. `obj` must be a single timer object.

`prop_cell=set(obj,'PropertyName')` returns the possible values for the specified property, `PropertyName`, of timer object `obj`. `obj` must be a single timer object. The returned array, `prop_cell`, is a cell array of possible value strings or an empty cell array if the property does not have a finite set of possible string values.

`set(obj,'PropertyName',PropertyValue,...)` configures the property, `PropertyName`, to the specified value, `PropertyValue`, for timer object `obj`. You can specify multiple property name/property value pairs in a single statement. `obj` can be a single timer object or a vector of timer objects, in which case `set` configures the property values for all the timer objects specified.
set (timer)

set(obj,S) configures the properties of obj, with the values specified in S, where S is a structure whose field names are object property names.

set(obj,PN,PV) configures the properties specified in the cell array of strings, PN, to the corresponding values in the cell array PV, for the timer object obj. PN must be a vector. If obj is an array of timer objects, PV can be an M-by-N cell array, where M is equal to the length of timer object array and N is equal to the length of PN. In this case, each timer object is updated with a different set of values for the list of property names contained in PN.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to set.

Examples

Create a timer object.

t = timer;

Display all configurable properties and their possible values.

set(t)

BusyMode: [ {drop} | queue | error ]
ErrorFcn: string -or- function handle -or- cell array
ExecutionMode: [ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
Name
ObjectVisibility: [ {on} | off ]
Period
StartDelay
StartFcn: string -or- function handle -or- cell array
StopFcn: string -or- function handle -or- cell array
Tag
TasksToExecute
TimerFcn: string -or- function handle -or- cell array
UserData

View the possible values of the ExecutionMode property.
set(t, 'ExecutionMode')
[ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]

Set the value of a specific timer object property.

set(t, 'ExecutionMode', 'FixedRate')

Set the values of several properties of the timer object.

set(t, 'TimerFcn', 'callbk', 'Period', 10)

Use a cell array to specify the names of the properties you want to set and another cell array to specify the values of these properties.

set(t, {'StartDelay', 'Period'}, {30, 30})

See Also

timer, get(timer)
### Purpose
Set properties of timeseries object

### Syntax
```
set(ts,'Property',Value)
set(ts,'Property1',Value1,'Property2',Value2,...)
set(ts,'Property')
set(ts)
```

### Description
- `set(ts,'Property',Value)` sets the property 'Property' of the timeseries object `ts` to the value `Value`. The following syntax is equivalent:
  ```
ts.Property = Value
  ```
- `set(ts,'Property1',Value1,'Property2',Value2,...)` sets multiple property values for `ts` with a single statement.
- `set(ts,'Property')` displays values for the specified property of the timeseries object `ts`.
- `set(ts)` displays all properties and values of the timeseries object `ts`.

### See Also
`get (timeseries)`
Purpose
Set properties of ts_collection object

Syntax
set(tsc,'Property',Value)
set(tsc,'Property1',Value1,'Property2',Value2,...)
set(tsc,'Property')

Description
set(tsc,'Property',Value) sets the property 'Property' of the ts_collection tsc to the value Value. The following syntax is equivalent:

    tsc.Property = Value

set(tsc,'Property1',Value1,'Property2',Value2,...) sets multiple property values for tsc with a single statement.

set(tsc,'Property') displays values for the specified property in the time-series collection tsc.

set(tsc) displays all properties and values of the ts_collection object tsc.

See Also
get (ts_collection)
Purpose
Set times of timeseries object as date strings

Syntax
\[
\text{ts} = \text{setabstime}(\text{ts}, \text{Times}) \\
\text{ts} = \text{setabstime}(\text{ts}, \text{Times}, \text{Format})
\]

Description
\[
\text{ts} = \text{setabstime}(\text{ts}, \text{Times}) \text{ sets the times in ts to the date strings specified in Times. Times must either be a cell array of strings, or a char array containing valid date or time values in the same date format.}
\]
\[
\text{ts} = \text{setabstime}(\text{ts}, \text{Times}, \text{Format}) \text{ explicitly specifies the date-string format used in Times.}
\]

Examples
1 Create a time-series object.
\[
\text{ts} = \text{timeseries}([\text{rand}(3,1)])
\]

2 Set the absolute time vector.
\[
\text{ts} = \text{setabstime}(\text{ts}, \{'12-DEC-2005 12:34:56', ... \text{'12-DEC-2005 14:34:56'}\})
\]

See Also
datestr, getabstime (timeseries), timeseries
Purpose

Set times of tscollection object as date strings

Syntax

tsc = setabstime(tsc,Times)
tsc = setabstime(tsc,Times,format)

Description

tsc = setabstime(tsc,Times) sets the times in tsc using the date strings Times. Times must be either a cell array of strings, or a char array containing valid date or time values in the same date format.

tsc = setabstime(tsc,Times,format) specifies the date-string format used in Times explicitly.

Examples

1 Create a tsollection object.

   tsc = tsollection(timeseries(rand(3,1)))

2 Set the absolute time vector.


See Also
datestr, getabstime (tscollection), tscollection
**Purpose**
Specify application-defined data

**Syntax**
setappdata(h,'name',value)

**Description**
setappdata(h,'name',value) sets application-defined data for the object with handle h. The application-defined data, which is created if it does not already exist, is assigned the specified name and value. The value can be any type of data.

**Remarks**
Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its AppData property. Only Handle Graphics MATLAB objects use this property.

**See Also**
getappdata, isappdata, rmappdata
**Purpose**
Set default random number stream

**Syntax**
prevstream = RandStream.setDefaultStream(stream)

**Description**
prevstream = RandStream.setDefaultStream(stream) returns the current default random number stream, and designates the random number stream stream as the new default to be used by the rand, randi, and randn functions.

rand, randi, and randn all rely on the same stream of uniform pseudorandom numbers, known as the default stream. randi uses one uniform value from the default stream to generate each integer value. randn uses one or more uniform values from the default stream to generate each normal value. Note that there are also rand, randi, and randn methods for which you specify a specific random stream from which to draw values.

**See Also**
getDefaultrandomstream (RandStream), @RandStream, rand (RandStream), randn (RandStream), randperm (RandStream)
Purpose
Find set difference of two vectors

Syntax
\[
\begin{align*}
c &= \text{setdiff}(A, B) \\
c &= \text{setdiff}(A, B, 'rows') \\
[c,i] &= \text{setdiff}(...) 
\end{align*}
\]

Description
\[
c = \text{setdiff}(A, B) \text{ returns the values in } A \text{ that are not in } B. \text{ In set theory terms, } c = A - B. \text{ Inputs } A \text{ and } B \text{ can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.}
\]
\[
c = \text{setdiff}(A, B, 'rows'), \text{ when } A \text{ and } B \text{ are matrices with the same number of columns, returns the rows from } A \text{ that are not in } B.
\]
\[
[c,i] = \text{setdiff}(...) \text{ also returns an index vector index such that } c = a(i) \text{ or } c = a(i,:).
\]

Remarks
Because NaN is considered to be not equal to itself, it is always in the result \(c\) if it is in \(A\).

Examples
\[
A = \text{magic}(5);
B = \text{magic}(4);
[c, i] = \text{setdiff}(A(:), B(:));
\]
\[
\begin{array}{cccccccccc}
c' &=& 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 \\
i' &=& 1 & 10 & 14 & 18 & 19 & 23 & 2 & 6 & 15
\end{array}
\]

See Also
intersect, ismember, issorted, setxor, union, unique
**Purpose**
Override to change command window display

**Syntax**
setdisp(H)

**Description**
setdisp(H) called by set when set is called with no output arguments and a single input argument that is a handle array. Override this hgsetget class method in a subclass to change how property information is displayed in the command window.

**See Also**
See “Implementing a Set/Get Interface for Properties”

hgsetget, set (hgsetget)
setenv

Purpose
Set environment variable

Syntax
setenv(name, value)
setenv(name)

Description
setenv(name, value) sets the value of an environment variable belonging to the underlying operating system. Inputs name and value are both strings. If name already exists as an environment variable, then setenv replaces its current value with the string given in value. If name does not exist, setenv creates a new environment variable called name and assigns value to it.

setenv(name) is equivalent to setenv(name, '') and assigns a null value to the variable name. On the Microsoft Windows platform, this is equivalent to undefining the variable. On most UNIX\textsuperscript{25} platforms, it is possible to have an environment variable defined as empty.

The maximum number of characters in name is $2^{15} - 2$ (or 32766). If name contains the character =, setenv throws an error. The behavior of environment variables with = in the name is not well-defined.

On all platforms, setenv passes the name and value strings to the operating system unchanged. Special characters such as ;, /, :, $, %, etc. are left unexpanded and intact in the variable value.

Values assigned to variables using setenv are picked up by any process that is spawned using the MATLAB system, unix, dos or ! functions. You can retrieve any value set with setenv by using getenv(name).

Examples
% Set and retrieve a new value for the environment variable TEMP:

$$\text{setenv('TEMP', 'C:\TEMP');}$$
$$\text{getenv('TEMP') }$$

% Append the Perl\bin directory to your system PATH variable:

25. UNIX is a registered trademark of The Open Group in the United States and other countries.
setenv('PATH', [getenv('PATH') ';D:\Perl\bin']);

**See Also**

getenv, system, unix, dos,
**Purpose**
Set value of structure array field

**Syntax**

```matlab
s = setfield(s, 'field', v)
s = setfield(s, {i,j}, 'field', {k}, v)
```

**Description**

`s = setfield(s, 'field', v)`, where `s` is a 1-by-1 structure, sets the contents of the specified field to the value `v`. If `field` is not an existing field in structure `s`, the MATLAB software creates that field and assigns the value `v` to it. This is equivalent to the syntax `s.field = v`.

`s = setfield(s, {i,j}, 'field', {k}, v)` sets the contents of the specified field to the value `v`. If `field` is not an existing field in structure `s`, MATLAB creates that field and assigns the value `v` to it. This is equivalent to the syntax `s(i,j).field(k) = v`. All subscripts must be passed as cell arrays — that is, they must be enclosed in curly braces (similar to `{i,j}` and `{k}` above). Pass field references as strings.

See “Guidelines for Naming Structure Fields” for help on creating valid field names.

**Remarks**
In many cases, you can use dynamic field names in place of the `getfield` and `setfield` functions. Dynamic field names express structure fields as variable expressions that MATLAB evaluates at run-time. See Solution 1-19QWG for information about using dynamic field names versus the `getfield` and `setfield` functions.

**Examples**
Given the structure

```matlab
myst(1,1).name = 'alice';
myst(1,1).ID = 0;
myst(2,1).name = 'gertrude';
myst(2,1).ID = 1;
```

You can change the `name` field of `myst(2,1)` using

```matlab
myst = setfield(myst, {2,1}, 'name', 'ted');
myst(2,1).name
```
ans =

ted

The following example sets fields of a structure using `setfield` with variable and quoted field names and additional subscripting arguments.

class = 5; student = 'John_Doe';
grades_Doe = [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
grades = [];

grades = setfield(grades, {class}, student, 'Math', ...
{10, 21:30}, grades_Doe);

You can check the outcome using the standard structure syntax.

grades(class).John_Doe.Math(10, 21:30)

ans =

85 89 76 93 85 91 68 84 95 73

See Also

`getfield`, `fieldnames`, `isfield`, `orderfields`, `rmfield`, `dynamic field names`
**Purpose**
Set default interpolation method for timeseries object

**Syntax**

```matlab
% setdefaultinterpolationmethod

ts = setinterpmethod(ts,Method)

% setdefaultinterpolationmethod with function handle

ts = setinterpmethod(ts,FHandle)

% setdefaultinterpolationmethod with interpolation object

ts = setinterpmethod(ts,InterpObj),
```

**Description**

`ts = setinterpmethod(ts,Method)` sets the default interpolation method for timeseries object `ts`, where `Method` is a string. `Method` in `ts` Method is either 'linear' or 'zoh' (zero-order hold). For example:

```matlab
% Set default interpolation method

ts = timeseries(rand(100,1),1:100);
% Set default interpolation method 'zoh'
ts = setinterpmethod(ts,'zoh');
```

`ts = setinterpmethod(ts,FHandle)` sets the default interpolation method for timeseries object `ts`, where FHandle is a function handle to the interpolation method defined by the function handle FHandle. For example:

```matlab
% Set default interpolation method with function handle

ts = timeseries(rand(100,1),1:100);
myFuncHandle = @(new_Time,Time,Data)...
    interp1(Time,Data,new_Time,...
    'linear','extrap');
% Set default interpolation method with function handle

ts = setinterpmethod(ts,myFuncHandle);
ts = resample(ts,[-5:0.1:10]);
plot(ts);
```

**Note** For FHandle, you must use three input arguments. The order of input arguments must be `new_Time`, `Time`, and `Data`. The single output argument must be the interpolated data only.

`ts = setinterpmethod(ts,InterpObj),` where `InterpObj` is a `tsdata.interpolation` object that directly replaces the interpolation object stored in `ts`. For example:

```matlab
% Set default interpolation method with interpolation object

ts = timeseries(rand(100,1),1:100);
```
myFuncHandle = @(new_Time,Time,Data)...
    interp1(Time,Data,new_Time,...
    'linear','extrap');
myInterpObj = tsdata.interpolation(myFuncHandle);
ts = setinterpmethod(ts,myInterpObj);

This method is case sensitive.

See Also
getinterpmethod, timeseries, tsprops
**setpixelposition**

**Purpose**
Set component position in pixels

**Syntax**

```matlab
setpixelposition(handle, position)
setpixelposition(handle, position, recursive)
```

**Description**

`setpixelposition(handle, position)` sets the position of the component specified by `handle`, to the specified pixel position relative to its parent. `position` is a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

`setpixelposition(handle, position, recursive)` sets the position as above. If `recursive` is true, the position is set relative to the parent figure of `handle`.

**Example**

This example first creates a push button within a panel.

```matlab
f = figure('Position', [300 300 300 200]);
p = uipanel('Position', [.2 .2 .6 .6];
h1 = uicontrol(p,'Style','PushButton','Units','Normalized',
    'String','Push Button','Position', [.1 .1 .5 .2]);
```
The example then retrieves the position of the push button and changes its position with respect to the panel.

```matlab
pos1 = getpixelposition(h1);
setpixelposition(h1, pos1 + [10 10 25 25]);
```
setpixelposition

See Also

getpixelposition, uicontrol, uipanel
Purpose

Set preference

Syntax

setpref('group','pref',val)
setpref('group',{'pref1','pref2',...,'prefn'},[val1,val2,...,valn])

Description

setpref('group','pref',val) sets the preference specified by group and pref to the value val. Setting a preference that does not yet exist causes it to be created.

group labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g., 'MathWorks_GUIDE_ApplicationPrefs'. The input argument pref identifies an individual preference in that group, and must be a legal variable name.

setpref('group',{'pref1','pref2',...,'prefn'},[val1,val2,...,valn]) sets each preference specified in the cell array of names to the corresponding value.

Note

Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

addpref('mytoolbox','version','0.0')
setpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')

ans =
1.0

See Also

addpref, getpref, ispref, rmpref, uigetpref, uisetpref
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Set string flag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>This MATLAB 4 function has been renamed <code>char</code> in MATLAB 5.</td>
</tr>
</tbody>
</table>
**Purpose**  
Change name of timeseries object in `tscollection`

**Syntax**  
`tsc = settimeseriesnames(tsc,old,new)`

**Description**  
`tsc = settimeseriesnames(tsc,old,new)` replaces the old name of timeseries object with the new name in `tsc`.

**See Also**  
`tscollection`
Purpose
Find set exclusive OR of two vectors

Syntax
c = setxor(A, B)
c = setxor(A, B, 'rows')
[c, ia, ib] = setxor(...)

Description
c = setxor(A, B) returns the values that are not in the intersection of A and B. Inputs A and B can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted.

c = setxor(A, B, 'rows'), when A and B are matrices with the same number of columns, returns the rows that are not in the intersection of A and B.

[c, ia, ib] = setxor(...) also returns index vectors ia and ib such that c is a sorted combination of the elements c = a(ia) and c = b(ib) or, for row combinations, c = a(ia,:) and c = b(ib,:).

Examples
a = [-1 0 1 Inf -Inf NaN];
b = [-2 pi 0 Inf];
c = setxor(a, b)

c =
    -Inf  -2.0000  -1.0000   1.0000   3.1416   NaN

See Also
intersect, ismember, issorted, setdiff, union, unique
### Purpose
Set color shading properties

### Syntax
- `shading flat`
- `shading faceted`
- `shading interp`
- `shading(axes_handle,...)`

### Description
The shading function controls the color shading of surface and patch graphics objects.

- `shading flat` each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.

- `shading faceted` flat shading with superimposed black mesh lines. This is the default shading mode.

- `shading interp` varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.

- `shading(axes_handle,...)` applies the shading type to the objects in the axes specified by `axes_handle`, instead of the current axes.

### Examples
Compare a flat, faceted, and interpolated-shaded sphere.

```matlab
subplot(3,1,1)
sphere(16)
axis square
shading flat
title('Flat Shading')

subplot(3,1,2)
sphere(16)
axis square
shading faceted
title('Faceted Shading')

subplot(3,1,3)
```
Algorithm

shading sets the `EdgeColor` and `FaceColor` properties of all surface and patch graphics objects in the current axes. `shading` sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

`fill`, `fill3`, `hidden`, `light`, `lighting`, `mesh`, `patch`, `pcolor`, `surf`

The `EdgeColor` and `FaceColor` properties for patch and surface graphics objects.

“Color Operations” on page 1-105 for related functions.
| **Purpose** | Show most recent graph window |
| **Syntax** | shg |
| **Description** | shg makes the current figure visible and raises it above all other figures on the screen. This is identical to using the command `figure(gca)`. |
| **See Also** | figure, gca, gcf |
Purpose
Shift dimensions

Syntax
B = shiftdim(X,n)
[B,nshifts] = shiftdim(X)

Description
B = shiftdim(X,n) shifts the dimensions of X by n. When n is positive, shiftdim shifts the dimensions to the left and wraps the n leading dimensions to the end. When n is negative, shiftdim shifts the dimensions to the right and pads with singletons.

[B,nshifts] = shiftdim(X) returns the array B with the same number of elements as X but with any leading singleton dimensions removed. A singleton dimension is any dimension for which size(A,dim) = 1. nshifts is the number of dimensions that are removed.

If X is a scalar, shiftdim has no effect.

Examples
The shiftdim command is handy for creating functions that, like sum or diff, work along the first nonsingleton dimension.

    a = rand(1,1,3,1,2);
    [b,n] = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
    c = shiftdim(b,-n); % c == a.
    d = shiftdim(a,3); % d is 1-by-2-by-1-by-1-by-3.

See Also
circshift, reshape, squeeze, permute, ipermute
**Purpose**

Show or hide figure plot tool

![Figure Plot Tool](image)

**GUI Alternatives**

Click the larger Plotting Tools icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Individually select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see “Plotting Tools — Interactive Plotting” in the MATLAB Graphics documentation.

**Syntax**

- `showplottool('tool')`
- `showplottool('on','tool')`
- `showplottool('off','tool')`
- `showplottool('toggle','tool')`
- `showplottool(figure_handle,...)`

**Description**

`showplottool('tool')` shows the specified plot tool on the current figure. `tool` can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`
showplottool('on','tool') shows the specified plot tool on the current figure.

showplottool('off','tool') hides the specified plot tool on the current figure.

showplottool('toggle','tool') toggles the visibility of the specified plot tool on the current figure.

showplottool(figure_handle,...) operates on the specified figure instead of the current figure.

**Note** When you dock, undock, resize, or reposition a plotting tool and then close it, it will still be configured as you left it the next time you open it. There is no command to reset plotting tools to their original, default locations.

**See Also**

figurepalette, plotbrowser, plottools, propertyeditor
Purpose

Reduce size of patch faces

Syntax

shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p)
nfv = shrinkfaces(f,v,sf)
[nf,nv] = shrinkfaces(...)

Description

shrinkfaces(p,sf) shrinks the area of the faces in patch p to shrink factor sf. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, the MATLAB software creates nonshared vertices before performing the face-area reduction.

nfv = shrinkfaces(p,sf) returns the face and vertex data in the struct nfv, but does not set the Faces and Vertices properties of patch p.

nfv = shrinkfaces(fv,sf) uses the face and vertex data from the struct fv.

shrinkfaces(p) and shrinkfaces(fv) (without specifying a shrink factor) assume a shrink factor of 0.3.

nfv = shrinkfaces(f,v,sf) uses the face and vertex data from the arrays f and v.

[nf,nv] = shrinkfaces(...) returns the face and vertex data in two separate arrays instead of a struct.

Examples

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type help flow for more information). Two isosurfaces provide a before and after view of the effects of shrinking the face size.

- First reducevolume samples the flow data at every other point and then isosurface generates the faces and vertices data.
The patch command accepts the face/vertex struct and draws the first (p1) isosurface.

Use the daspect, view, and axis commands to set up the view and then add a title.

The shrinkfaces command modifies the face/vertex data and passes it directly to patch.

```matlab
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
set(p1,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('Original')

figure
p2 = patch(shrinkfaces(fv,.3));
set(p2,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('After Shrinking')
```
shrinkfaces

See Also

isosurface, patch, reducevolume, daspect, view, axis

“Volume Visualization” on page 1-108 for related functions
Purpose

Signum function

Syntax

Y = sign(X)

Description

Y = sign(X) returns an array Y the same size as X, where each element of Y is:

- 1 if the corresponding element of X is greater than zero
- 0 if the corresponding element of X equals zero
- -1 if the corresponding element of X is less than zero

For nonzero complex X, sign(X) = X./abs(X).

See Also

abs, conj, imag, real
**Purpose**  
Sine of argument in radians

**Syntax**  
\[ Y = \sin(X) \]

**Description**  
The \( \sin \) function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians. \( Y = \sin(X) \) returns the circular sine of the elements of \( X \).

**Examples**  
Graph the sine function over the domain \( -\pi \leq x \leq \pi \).

\[
x = -\pi:0.01:pi;
\text{plot}(x,\sin(x)), \text{grid on}
\]

The expression \( \sin(\pi) \) is not exactly zero, but rather a value the size of the floating-point accuracy \( \text{eps} \), because \( \pi \) is only a floating-point approximation to the exact value of \( \pi \).
**Definition**

The sine can be defined as

$$\sin(x + iy) = \sin(x)\cosh(y) + i\cos(x)\sinh(y)$$

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}$$

**Algorithm**

`sin` uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

**See Also**

`sind`, `sinh`, `asin`, `asind`, `asinh`
**Purpose**  
Sine of argument in degrees

**Syntax**  
\[ Y = \text{sind}(X) \]

**Description**  
\( Y = \text{sind}(X) \) is the sine of the elements of \( X \), expressed in degrees. For integers \( n \), \( \text{sind}(n\times180) \) is exactly zero, whereas \( \sin(n\times\pi) \) reflects the accuracy of the floating point value of \( \pi \).

**See Also**  
\( \text{sin, sinh, asin, asind, asinh} \)
Purpose
Convert to single precision

Syntax
B = single(A)

Description
B = single(A) converts the matrix A to single precision, returning that value in B. A can be any numeric object (such as a double). If A is already single precision, single has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The single class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are reshape, size, the relational operators, subscripted assignment, and subscripted reference.

You can define your own methods for the single class by placing the appropriately named method in an @single directory within a directory on your path.

Examples
a = magic(4);
b = single(a);

whos
Name        Size     Bytes  Class
----        -----     -----  -----
a          4x4       128    double array
b          4x4        64    single array

See Also
double
**Purpose**
Hyperbolic sine of argument in radians

**Syntax**

\[ Y = \sinh(X) \]

**Description**
The \( \sinh \) function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

\[ Y = \sinh(X) \]
returns the hyperbolic sine of the elements of \( X \).

**Examples**
Graph the hyperbolic sine function over the domain \( -5 \leq x \leq 5 \).

\[
x = -5:0.01:5;
\]
\[
plot(x,\sinh(x)), \text{ grid on}
\]

**Definition**
The hyperbolic sine can be defined as

\[
\sinh(z) = \frac{e^z - e^{-z}}{2}
\]
**Algorithm**  
sinh uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

**See Also**  
sin, sind, asin, asinh, asind
**Purpose**
Array dimensions

**Syntax**

\[
d = \text{size}(X) \\
[m,n] = \text{size}(X) \\
m = \text{size}(X,\text{dim}) \\
[d1,d2,d3,...,dn] = \text{size}(X),
\]

**Description**

\( d = \text{size}(X) \) returns the sizes of each dimension of array \( X \) in a vector \( d \) with \( \text{ndims}(X) \) elements. If \( X \) is a scalar, which MATLAB software regards as a 1-by-1 array, \( \text{size}(X) \) returns the vector \([1 \ 1]\).

\([m,n] = \text{size}(X)\) returns the size of matrix \( X \) in separate variables \( m \) and \( n \).

\( m = \text{size}(X,\text{dim}) \) returns the size of the dimension of \( X \) specified by scalar \( \text{dim} \).

\([d1,d2,d3,...,dn] = \text{size}(X), \text{ for } n > 1, \) returns the sizes of the dimensions of the array \( X \) in the variables \( d1,d2,d3,...,dn \), provided the number of output arguments \( n \) equals \( \text{ndims}(X) \). If \( n \) does not equal \( \text{ndims}(X) \), the following exceptions hold:

\[ n < \text{ndims}(X) \quad \text{di equals the size of the ith dimension of } X \text{ for } 1 \leq i < n, \] but \( dn \) equals the product of the sizes of the remaining dimensions of \( X \), that is, dimensions \( n \) through \( \text{ndims}(X) \).

\[ n > \text{ndims}(X) \quad \text{size returns ones in the “extra” variables, that is, those corresponding to } \text{ndims}(X)+1 \text{ through } n. \]

**Note**
For a Java array, \( \text{size} \) returns the length of the Java array as the number of rows. The number of columns is always 1. For a Java array of arrays, the result describes only the top level array.

**Examples**

**Example 1**

The size of the second dimension of \( \text{rand}(2,3,4) \) is 3.
m = size(rand(2,3,4),2)

m =

3

Here the size is output as a single vector.

d = size(rand(2,3,4))

d =

2 3 4

Here the size of each dimension is assigned to a separate variable.

[m,n,p] = size(rand(2,3,4))

m =

2

n =

3

p =

4

Example 2

If \( X = \text{ones}(3,4,5) \), then

\[
[d1,d2,d3] = \text{size}(X)
\]

\[
\begin{align*}
d1 &= 3 \\
d2 &= 4 \\
d3 &= 5
\end{align*}
\]

But when the number of output variables is less than \text{nndims}(X):

\[
[d1,d2] = \text{size}(X)
\]

\[
\begin{align*}
d1 &= 3 \\
d2 &= 20
\end{align*}
\]
The “extra” dimensions are collapsed into a single product.
If \( n > \text{ndims}(X) \), the “extra” variables all represent singleton dimensions:

\[
[d_1, d_2, d_3, d_4, d_5, d_6] = \text{size}(X)
\]

\[
d_1 = 3, \quad d_2 = 4, \quad d_3 = 5
\]

\[
d_4 = 1, \quad d_5 = 1, \quad d_6 = 1
\]

**See Also**  
exist, length, numel, whos
**Purpose**
size of containers.Map object

**Syntax**
d = size(M)
d = size(M, dim)
[d1, d2, ..., dn] = size(M)

**Description**
d = size(M) returns the number of key-value pairs in dimensions 1 and 2 of map M. Output d is a two-element row vector [n, 1], where n is the number of key-value pairs.

d = size(M, dim) returns the number of key-value pairs if dim is 1, and otherwise returns 1.

[d1, d2, ..., dn] = size(M) returns [n, 1, ..., 1] where n is the number of key-value pairs in map M.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

**Examples**
Create a Map object containing the names of several US states and the capital city of each:

```matlab
US_Capitals = containers.Map( ... 
    {'Arizona', 'Nebraska', 'Nevada', 'New York', ... 
    'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ... 
    {'Phoenix', 'Lincoln', 'Carson City', 'Albany', ... 
    'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Get the dimensions of the Map object array:

```matlab
size(US_Capitals)
ans =
     8     1
```

Use the map to find the capital of one of these states:

```matlab
state = 'Georgia';
sprintf(' The capital of %s is %s', ...
    state, US_Capitals(state))
```
size (Map)

ans =
    The capital of Georgia is Atlanta

See Also

containers.Map, keys(Map), values(Map), length(Map), isKey(Map), remove(Map), handle
**Purpose**
Size of serial port object array

**Syntax**

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

**Description**

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in the serial port object, `obj`.

`[m,n] = size(obj)` returns the number of rows, `m` and columns, `n` in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

**See Also**

**Functions**

length
**size (timeseries)**

**Purpose**  
Size of timeseries object

**Syntax**  
size(ts)

**Description**  
size(ts) returns \([n \ 1]\), where \(n\) is the length of the time vector for timeseries object ts.

**Remarks**  
If you want the size of the whole data set, use the following syntax:

```
size(ts.data)
```

If you want the size of each data sample, use the following syntax:

```
getdatasamplesize(ts)
```

**See Also**  
getdatasamplesize, isempty (timeseries), length (timeseries)
**Purpose**  
Size of triangulation matrix

**Syntax**  
size(TR)

**Description**  
size(TR) provides size information for a triangulation matrix. The matrix is of size $m_{tri}$-by-$nv$, where $m_{tri}$ is the number of simplices and $nv$ is the number of vertices per simplex (triangle/tetrahedron, etc).

**Inputs**  
TR  
Triangulation matrix

**Definitions**  
A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

**See Also**  
size  
“Triangulation Representations”—How to query triangulation data.
size (tscollection)

**Purpose**  Size of tscollection object

**Syntax**  size(tsc)

**Description**  size(tsc) returns [n m], where n is the length of the time vector and m is the number of tscollection members.

**See Also**  length (tscollection), isempty (tscollection), tscollection
Volumetric slice plot

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

### Syntax

- `slice(V,sx,sy,sz)`
- `slice(X,Y,Z,V,sx,sy,sz)`
- `slice(V,XI,YI,ZI)`
- `slice(X,Y,Z,V,XI,YI,ZI)`
- `slice(...,'method')`
- `slice(axes_handle,...)`
- `h = slice(...)`

### Description

*slice* displays orthogonal slice planes through volumetric data.

*`slice(V,sx,sy,sz)`* draws slices along the *x*, *y*, *z* directions in the volume *V* at the points in the vectors *sx*, *sy*, and *sz*. *V* is an *m*-by-*n*-by-*p* volume array containing data values at the default location *X* = 1:*n*, *Y* = 1:*m*, *Z* = 1:*p*. Each element in the vectors *sx*, *sy*, and *sz* defines a slice plane in the *x*-, *y*-, or *z*-axis direction.

*`slice(X,Y,Z,V,sx,sy,sz)`* draws slices of the volume *V*. *X*, *Y*, and *Z* are three-dimensional arrays specifying the coordinates for *V*. *X*, *Y*, and *Z* must be monotonic and orthogonally spaced (as if produced by the function *meshgrid*). The color at each point is determined by 3-D interpolation into the volume *V*.

*`slice(V,XI,YI,ZI)`* draws data in the volume *V* for the slices defined by *XI*, *YI*, and *ZI*. *XI*, *YI*, and *ZI* are matrices that define a surface, and the volume is evaluated at the surface points. *XI*, *YI*, and *ZI* must all be the same size.
slice(X,Y,Z,V,XI,YI,ZI) draws slices through the volume V along
the surface defined by the arrays XI, YI, ZI.

slice(...,'method') specifies the interpolation method. 'method' is
'linear', 'cubic', or 'nearest'.

- **linear** specifies trilinear interpolation (the default).
- **cubic** specifies tricubic interpolation.
- **nearest** specifies nearest-neighbor interpolation.

slice(axes_handle,...) plots into the axes with the handle
axes_handle instead of into the current axes object (gca). The axes
clim property is set to span the finite values of V.

h = slice(...) returns a vector of handles to surface graphics
objects.

**Remarks**

The color drawn at each point is determined by interpolation into the
volume V.

**Examples**

Visualize the function

\[ v = xe^{-x^2-y^2-z^2} \]

over the range \(-2 \leq x \leq 2, -2 \leq y \leq 2, -2 \leq z \leq 2:\)

\[
[x,y,z] = \text{meshgrid}(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
xslice = [-1.2,.8,2]; yslice = 2; zslice = [-2,0];
slice(x,y,z,v,xslice,yslice,zslice)
colormap hsv\]
Slicing At Arbitrary Angles

You can also create slices that are oriented in arbitrary planes. To do this,

- Create a slice surface in the domain of the volume (surf, linspace).
- Orient this surface with respect to the axes (rotate).
- Get the XData, YData, and ZData of the surface (get).
- Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a for loop “passes” the plane through the volume along the z-axis.

```matlab
for i = -2:.5:2
    hsp = surf(linspace(-2,2,20),linspace(-2,2,20),zeros(20)+i);
```
rotate(hsp,[1,-1,1],30)
xd = get(hsp,'XData');
yd = get(hsp,'YData');
zd = get(hsp,'ZData');
delete(hsp)
slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries
hold on
slice(x,y,z,v,xd,yd,zd)
hold off
axis tight
view(-5,10)
drawnow
end

The following picture illustrates three positions of the same slice surface as it passes through the volume.
Slicing with a Nonplanar Surface

You can slice the volume with any surface. This example probes the volume created in the previous example by passing a spherical slice surface through the volume.

```
[xsp, ysp, zsp] = sphere;
slice(x, y, z, v, [-2, 2], 2, -2) % Draw some volume boundaries

for i = -3:.2:3
    hsp = surface(xsp+i, ysp, zsp);
    rotate(hsp, [1 0 0], 90)
    xd = get(hsp, 'XData');
    yd = get(hsp, 'YData');
    zd = get(hsp, 'ZData');
    delete(hsp)
    hold on
    hslicer = slice(x, y, z, v, xd, yd, zd);
    axis tight
    xlim([-3, 3])
    view(-10, 35)
    drawnow
    delete(hslicer)
    hold off
end
```

The following picture illustrates three positions of the spherical slice surface as it passes through the volume.
See Also

interp3, meshgrid

“Volume Visualization” on page 1-108 for related functions

Exploring Volumes with Slice Planes for more examples
Purpose
Smooth 3-D data

Syntax

Description
$W = \text{smooth3}(V)$ smooths the input data $V$ and returns the smoothed data in $W$.

$W = \text{smooth3}(V, 'filter')$ $filter$ determines the convolution kernel and can be the strings

- 'gaussian'
- 'box' (default)

$W = \text{smooth3}(V, 'filter', size)$ sets the size of the convolution kernel (default is [3 3 3]). If $size$ is scalar, then $size$ is interpreted as $[size, size, size]$.

$W = \text{smooth3}(V, 'filter', size, sd)$ sets an attribute of the convolution kernel. When $filter$ is gaussian, $sd$ is the standard deviation (default is .65).

Examples
This example smooths some random 3-D data and then creates an isosurface with end caps.

```matlab
rand('seed',0)
data = rand(10,10,10);
data = smooth3(data, 'box', 5);
p1 = patch(isosurface(data, .5), ...  'FaceColor', 'blue', 'EdgeColor', 'none');
p2 = patch(isocaps(data, .5), ...  'FaceColor', 'interp', 'EdgeColor', 'none');
isonormals(data, p1)
view(3); axis vis3d tight
camlight; lighting phong
```
See Also

isocaps, isonormals, isosurface, patch

“Volume Visualization” on page 1-108 for related functions

See Displaying an Isosurface for another example.
**Purpose**

Force snapshot of image for inclusion in published document

**GUI Alternative**

As an alternative to `snapnow`, open an M-File and select **Cell > Insert Text Markup > Force Snapshot** to insert the `snapnow` command into the M-File.

**Syntax**

`snapnow`

**Description**

The `snapnow` command forces a snapshot of the image or plot that the code has most recently generated for presentation in a published document. The output appears in the published document at the end of the cell that contains the `snapnow` command. When used outside the context of publishing an M-File, `snapnow` has the same behavior as `drawnow`. That is, if you run a file that contains the `snapnow` command, the MATLAB software interprets it as though it were a `drawnow` command.

**Example**

This example demonstrates the difference between publishing code that contains the `snapnow` command and running that code. The first image shows the results of publishing the code and the second image shows the results of running the code.

Suppose you have an M-file that contains the following code:

```plaintext
%% Scale magic Data and
%% Display as Image:

for i=1:3
    i
    imagesc(magic(i))
    snapnow
end
```

When you publish the code to HTML, the published document contains a title, a table of contents, the commented text, the code, and each of the three images produced by the `for` loop, along with a display of the value 2-3305.
of i corresponding to each image. (In the published document shown, the size of the images have been reduced.)
Scale magic Data and

Contents

- Display as Image

Display as Image:

```matlab
for i=1:3
    i
    imagesc(magic(i))
    snapnow
end
```

```matlab
i =

1
```

```matlab
i =

2
```

```matlab
i =

3
```
When you run the code, a single Figure window opens and MATLAB updates the image within this window as it evaluates each iteration of the `for` loop. (Concurrently, the Command Window displays the value of `i`.) Each successive image replaces the one that preceded it, so that the Figure window appears as follows when the code evaluation completes.
See Also

drawnow

“Forcing a Snapshot of Output in M-Files for Publishing”
**Purpose**
Sort array elements in ascending or descending order

**Syntax**

\[ B = \text{sort}(A) \]
\[ B = \text{sort}(A,\text{dim}) \]
\[ B = \text{sort}(\ldots,\text{mode}) \]
\[ [B,IX] = \text{sort}(A,\ldots) \]

**Description**

\( B = \text{sort}(A) \) sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

<table>
<thead>
<tr>
<th>If A is a ...</th>
<th>sort(A) ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>Sorts the elements of A.</td>
</tr>
<tr>
<td>Matrix</td>
<td>Sorts each column of A.</td>
</tr>
<tr>
<td>Multidimensional array</td>
<td>Sorts A along the first non-singleton dimension, and returns an array of sorted vectors.</td>
</tr>
<tr>
<td>Cell array of strings</td>
<td>Sorts the strings in ASCII dictionary order.</td>
</tr>
</tbody>
</table>

Integer, floating-point, logical, and character arrays are permitted. Floating-point arrays can be complex. For elements of A with identical values, the order of these elements is preserved in the sorted list. When A is complex, the elements are sorted by magnitude, i.e., \( \text{abs}(A) \), and where magnitudes are equal, further sorted by phase angle, i.e., \( \text{angle}(A) \), on the interval \([-\pi, \pi]\). If A includes any NaN elements, sort places these at the high end.

\( B = \text{sort}(A,\text{dim}) \) sorts the elements along the dimension of A specified by a scalar \( \text{dim} \).

\( B = \text{sort}(\ldots,\text{mode}) \) sorts the elements in the specified direction, depending on the value of \( \text{mode} \).

- `'ascend'` Ascending order (default)
- `'descend'` Descending order
[B,IX] = sort(A,...) also returns an array of indices IX, where size(IX) == size(A). If A is a vector, B = A(IX). If A is a m-by-n matrix, then each column of IX is a permutation vector of the corresponding column of A, such that

```matlab
for j = 1:n
    B(:,j) = A(IX(:,j),j);
end
```

If A has repeated elements of equal value, the returned indices preserve the original ordering.

**Sorting Complex Entries**

If A has complex entries r and s, sort orders them according to the following rule: r appears before s in sort(A) if either of the following hold:

- abs(r) < abs(s)
- abs(r) = abs(s) and angle(r)<angle(s)

where $-\pi < angle(r) \leq \pi$

For example,

```matlab
v = [1 -1 i -i];
angle(v)
```

```matlab
ans =
    0     3.1416    1.5708   -1.5708
```

```matlab
sort(v)
```

```matlab
ans =
    0 - 1.0000i    1.0000
    0 + 1.0000i   -1.0000
```
Note sort uses a different rule for ordering complex numbers than do the relational operators. See the Relational Operators reference page for more information. For more information about how MATLAB software treats complex numbers, see “Numbers” in the MATLAB Getting Started Guide.

Examples

Example 1

This example sorts a matrix A in each dimension, and then sorts it a third time, returning an array of indices for the sorted result.

```matlab
A = [ 3 7 5
     0 4 2 ];

sort(A,1)

ans =
     0  4  2
     3  7  5

sort(A,2)

ans =
     3  5  7
     0  2  4

[B,IX] = sort(A,2)

B =
     3  5  7
     0  2  4

IX =
     1  3  2
     1  3  2
```
Example 2

This example sorts each column of a matrix in descending order.

\[
A = \begin{bmatrix}
3 & 7 & 5 \\
6 & 8 & 3 \\
0 & 4 & 2 \\
\end{bmatrix};
\]

\[
\text{sort}(A,1,'\text{descend}')
\]

\[
\text{ans} =
\begin{bmatrix}
6 & 8 & 5 \\
3 & 7 & 3 \\
0 & 4 & 2 \\
\end{bmatrix}
\]

This is equivalent to

\[
\text{sort}(A,'\text{descend}')
\]

\[
\text{ans} =
\begin{bmatrix}
6 & 8 & 5 \\
3 & 7 & 3 \\
0 & 4 & 2 \\
\end{bmatrix}
\]

See Also

issorted, max, mean, median, min, sortrows, unique
Purpose
Sort rows in ascending order

Syntax
B = sortrows(A)
B = sortrows(A,column)
[B,index] = sortrows(A,...)

Description
B = sortrows(A) sorts the rows of A in ascending order. Argument A must be either a matrix or a column vector.

For strings, this is the familiar dictionary sort. When A is complex, the elements are sorted by magnitude, and, where magnitudes are equal, further sorted by phase angle on the interval \([-\pi, \pi]\).

B = sortrows(A,column) sorts the matrix based on the columns specified in the vector column. If an element of column is positive, the MATLAB software sorts the corresponding column of matrix A in ascending order; if an element of column is negative, MATLAB sorts the corresponding column in descending order. For example, sortrows(A,[2 -3]) sorts the rows of A first in ascending order for the second column, and then by descending order for the third column.

[B,index] = sortrows(A,...) also returns an index vector index.

If A is a column vector, then B = A(index). If A is an m-by-n matrix, then B = A(index,:).

Examples
Start with a mostly random matrix, A:

```matlab
rand('state',0)
A = floor(rand(6,7) * 100);
A(1:4,1)=95; A(5:6,1)=76; A(2:4,2)=7; A(3,3)=73
A =
95  45  92  41  13  1  84
95   7  73  89  20  74  52
95   7  73   5  19  44  20
95   7  40  35  60  93  67
76  61  93  81  27  46  83
76  79  91  40  19  41  1
```

2-3314
When called with only a single input argument, `sortrows` bases the sort on the first column of the matrix. For any rows that have equal elements in a particular column, (e.g., `A(1:4,1)` for this matrix), sorting is based on the column immediately to the right, (`A(1:4,2)` in this case):

```plaintext
B = sortrows(A)
B=
    76   61   93   81   27   46   83
    76   79   91   0   19   41   1
    95    7   40   35   60   93   67
    95    7   73    5   19   44   20
    95    7   73   89   20   74   52
    95   45   92   41   13    1   84
```

When called with two input arguments, `sortrows` bases the sort entirely on the column specified in the second argument. Rows that have equal elements in the specified column, (e.g., `A(2:4,:)`, if sorting matrix `A` by column 2) remain in their original order:

```plaintext
C = sortrows(A,2)
C=
    95    7   73   89   20   74   52
    95    7   73    5   19   44   20
    95    7   40   35   60   93   67
    95   45   92   41   13    1   84
    76   61   93   81   27   46   83
    76   79   91    0   19   41    1
```

This example specifies two columns to sort by: columns 1 and 7. This tells `sortrows` to sort by column 1 first, and then for any rows with equal values in column 1, to sort by column 7:

```plaintext
D = sortrows(A,[1 7])
D=
    76   79   91    0   19   41   1
    76   61   93   81   27   46   83
    95    7   73    5   19   44   20
    95    7   73   89   20   74   52
```
Sort the matrix using the values in column 4 this time and in reverse order:

\[
E = \text{sortrows}(A, \ -4)
\]

\[
E =
\begin{bmatrix}
95 & 7 & 73 & 89 & 20 & 74 & 52 \\
76 & 61 & 93 & 81 & 27 & 46 & 83 \\
95 & 45 & 92 & 41 & 13 & 1 & 84 \\
95 & 7 & 40 & 35 & 60 & 93 & 67 \\
95 & 7 & 73 & 5 & 19 & 44 & 20 \\
76 & 79 & 91 & 0 & 19 & 41 & 1
\end{bmatrix}
\]

**See Also**

issorted, sort
Purpose
Convert vector into sound

Syntax
sound(y,Fs)
sound(y)
sound(y,Fs,bits)

Description
sound(y,Fs) sends the signal in vector y (with sample frequency Fs, in hertz) to the speaker on Microsoft Windows and most UNIX platforms. Values in y are assumed to be in the range $-1.0 \leq y \leq 1.0$. Values outside that range are clipped. Stereo sound is played on platforms that support it when y is an n-by-2 matrix. The values in column 1 are assigned to the left channel, and those in column 2 to the right.

Note
The playback duration that results from setting Fs depends on the sound card you have installed. Most sound cards support sample frequencies in the range of approximately 5 kHz to 44.1 kHz. Sample frequencies outside this range can produce unexpected results.

sound(y) plays the sound at the default sample rate or 8192 Hz.
sound(y,Fs,bits) plays the sound using bits number of bits/sample, if possible. Most platforms support bits = 8 or bits = 16.

Remarks
MATLAB software supports all sound devices compatible with Windows operating systems. Additional sound acquisition and generation capability is available in Data Acquisition Toolbox software. The toolbox functionality includes the ability to buffer the acquisition so that you can analyze the data as it is being acquired. See the examples on MATLAB sound acquisition and sound generation.

See Also
auread, auwrite, soundsc, audioplayer, wavread, wavwrite
Purpose

Scale data and play as sound

Syntax

soundsc(y,Fs)
soundsc(y)
soundsc(y,Fs,bits)
soundsc(y,...,slim)

Description

soundsc(y,Fs) sends the signal in vector y (with sample frequency Fs) to the speaker on PC and most UNIX platforms. The signal y is scaled to the range \(-1.0 \leq y \leq 1.0\) before it is played, resulting in a sound that is played as loud as possible without clipping.

Note

The playback duration that results from setting Fs depends on the sound card you have installed. Most sound cards support sample frequencies of approximately 5-10 kHz to 44.1 kHz. Sample frequencies outside this range can produce unexpected results.

soundsc(y) plays the sound at the default sample rate or 8192 Hz.
soundsc(y,Fs,bits) plays the sound using bits number of bits/sample if possible. Most platforms support bits = 8 or bits = 16.
soundsc(y,...,slim), where slim = [slow shigh], maps the values in y between slow and shigh to the full sound range. The default value is slim = [min(y) max(y)].

Remarks

MATLAB software supports all Windows-compatible sound devices.

See Also

auread, auwrite, sound, wavread, wavwrite
Purpose
Allocate space for sparse matrix

Syntax
S = spalloc(m,n,nzmax)

Description
S = spalloc(m,n,nzmax) creates an all zero sparse matrix S of size m-by-n with room to hold nzmax nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

spalloc(m,n,nzmax) is shorthand for

spare([],[],[],m,n,nzmax)

Examples
To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

S = spalloc(n,n,3*n);
for j = 1:n
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';end
Purpose
Create sparse matrix

Syntax

S = sparse(A)
S = sparse(i,j,s,m,n,nzmax)
S = sparse(i,j,s,m,n)
S = sparse(i,j,s)
S = sparse(m,n)

Description
The sparse function generates matrices in the MATLAB sparse storage organization.

S = sparse(A) converts a full matrix to sparse form by squeezing out any zero elements. If S is already sparse, sparse(S) returns S.

S = sparse(i,j,s,m,n,nzmax) uses vectors i, j, and s to generate an m-by-n sparse matrix such that S(i(k),j(k)) = s(k), with space allocated for nzmax nonzeros. Vectors i, j, and s are all the same length. Any elements of s that are zero are ignored, along with the corresponding values of i and j. Any elements of s that have duplicate values of i and j are added together.

Note If any value in i or j is larger than the maximum integer size, \(2^{31}-1\), then the sparse matrix cannot be constructed.

To simplify this six-argument call, you can pass scalars for the argument s and one of the arguments i or j—in which case they are expanded so that i, j, and s all have the same length.

S = sparse(i,j,s,m,n) uses nzmax = length(s).

S = sparse(i,j,s) uses \(m = \max(i)\) and \(n = \max(j)\). The maxima are computed before any zeros in s are removed, so one of the rows of [i j s] might be [m n 0].

S = sparse(m,n) abbreviates sparse([],[],[],m,n,0). This generates the ultimate sparse matrix, an m-by-n all zero matrix.
Remarks

All of the MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse, for example, A.*S is at least as sparse as S.

Examples

S = sparse(1:n,1:n,1) generates a sparse representation of the n-by-n identity matrix. The same S results from S = sparse(eye(n,n)), but this would also temporarily generate a full n-by-n matrix with most of its elements equal to zero.

B = sparse(10000,10000,pi) is probably not very useful, but is legal and works; it sets up a 10000-by-10000 matrix with only one nonzero element. Don’t try full(B); it requires 800 megabytes of storage.

This dissects and then reassembles a sparse matrix:

\[
[i,j,s] = \text{find}(S);
[m,n] = \text{size}(S);
S = \text{sparse}(i,j,s,m,n);
\]

So does this, if the last row and column have nonzero entries:

\[
[i,j,s] = \text{find}(S);
S = \text{sparse}(i,j,s);
\]

See Also

diag, find, full, issparse, nnz, nonzeros, nzmax, spones, sprandn, sprandsym, spy

The sparfuns directory
**Purpose**
Form least squares augmented system

**Syntax**

```
S = spaugment(A,c)
S = spaugment(A)
```

**Description**

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix `S = [c*I A; A' 0]`. The matrix `S` is related to the least squares problem

\[
\min \: \text{norm}(b - A*x)
\]

by

\[
\begin{align*}
  r &= b - A*x \\
  S * [r/c; x] &= [b; 0]
\end{align*}
\]

The optimum value of the residual scaling factor `c`, involves `\text{min(svd}(A))` and `\text{norm}(r)`, which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses `max(max(abs(A)))/1000`.

**Note** In previous versions of MATLAB product, the augmented matrix was used by sparse linear equation solvers, \ and /, for nonsquare problems. Now, MATLAB software performs a least squares solve using the qr factorization of `A` instead.

**See Also**

spparms
Purpose
Import matrix from sparse matrix external format

Syntax
S = spconvert(D)

Description
spconvert is used to create sparse matrices from a simple sparse
format easily produced by non-MATLAB sparse programs. spconvert
is the second step in the process:

1 Load an ASCII data file containing [i,j,v] or [i,j,re,im] as rows
into a MATLAB variable.

2 Convert that variable into a MATLAB sparse matrix.

S = spconvert(D) converts a matrix D with rows containing [i,j,s]
or [i,j,r,s] to the corresponding sparse matrix. D must have an
nnz or nnz+1 row and three or four columns. Three elements per row
generate a real matrix and four elements per row generate a complex
matrix. A row of the form [m n 0] or [m n 0 0] anywhere in D can be
used to specify size(S). If D is already sparse, no conversion is done, so
spconvert can be used after D is loaded from either a MAT-file or an
ASCII file.

Examples
Suppose the ASCII file uphill.dat contains

1 1 1.000000000000000
1 2 0.500000000000000
2 2 0.333333333333333
1 3 0.333333333333333
2 3 0.250000000000000
3 3 0.200000000000000
1 4 0.250000000000000
2 4 0.200000000000000
3 4 0.166666666666667
4 4 0.142857142857143
4 4 0.000000000000000

Then the statements
load uphill.dat
H = spconvert(uphill)

H =
(1,1) 1.0000  
(1,2) 0.5000  
(2,2) 0.3333  
(1,3) 0.3333  
(2,3) 0.2500  
(3,3) 0.2000  
(1,4) 0.2500  
(2,4) 0.2000  
(3,4) 0.1667  
(4,4) 0.1429

recreate sparse(triu(hilb(4))), possibly with roundoff errors. In this case, the last line of the input file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.
Purpose

Extract and create sparse band and diagonal matrices

Syntax

B = spdiags(A)
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)

Description

The spdiags function generalizes the function diag. Four different operations, distinguished by the number of input arguments, are possible.

B = spdiags(A) extracts all nonzero diagonals from the m-by-n matrix A. B is a min(m,n)-by-p matrix whose columns are the p nonzero diagonals of A.

[B,d] = spdiags(A) returns a vector d of length p, whose integer components specify the diagonals in A.

B = spdiags(A,d) extracts the diagonals specified by d.

A = spdiags(B,d,A) replaces the diagonals specified by d with the columns of B. The output is sparse.

A = spdiags(B,d,m,n) creates an m-by-n sparse matrix by taking the columns of B and placing them along the diagonals specified by d.

Note

In this syntax, if a column of B is longer than the diagonal it is replacing, and m >= n, spdiags takes elements of super-diagonals from the lower part of the column of B, and elements of sub-diagonals from the upper part of the column of B. However, if m < n, then super-diagonals are from the upper part of the column of B, and sub-diagonals from the lower part. (See “Example 5A” on page 2-3331 and “Example 5B” on page 2-3333, below).

Arguments

The spdiags function deals with three matrices, in various combinations, as both input and output.
A An m-by-n matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on p diagonals.

B A \min(m,n)\times p matrix, usually (but not necessarily) full, whose columns are the diagonals of A.

d A vector of length p whose integer components specify the diagonals in A.

Roughly, A, B, and d are related by

```matlab
for k = 1:p
    B(:,k) = diag(A,d(k))
end
```

Some elements of B, corresponding to positions outside of A, are not defined by these loops. They are not referenced when B is input and are set to zero when B is output.

**How the Diagonals of A are Listed in the Vector d**

An m-by-n matrix A has m+n-1 diagonals. These are specified in the vector d using indices from -m+1 to n-1. For example, if A is 5-by-6, it has 10 diagonals, which are specified in the vector d using the indices -4, -3, ... 4, 5. The following diagram illustrates this for a vector of all ones.
**Examples**

**Example 1**

For the following matrix,

\[
A = \begin{bmatrix}
0 & 5 & 0 & 10 & 0 & 0 \\
0 & 0 & 6 & 0 & 11 & 0 \\
3 & 0 & 0 & 7 & 0 & 12 \\
1 & 4 & 0 & 0 & 8 & 0 \\
0 & 2 & 5 & 0 & 0 & 9
\end{bmatrix}
\]

the command

\[
[B, d] = \text{spdiags}(A)
\]

returns

\[
B = \begin{bmatrix}
0 & 0 & 5 & 10 \\
0 & 0 & 6 & 11 \\
0 & 3 & 7 & 12 \\
1 & 4 & 8 & 0 \\
2 & 5 & 9 & 0
\end{bmatrix}
\]

\[
d = \\
-3 \\
-2 \\
1
\]
The columns of the first output $B$ contain the nonzero diagonals of $A$. The second output $d$ lists the indices of the nonzero diagonals of $A$, as shown in the following diagram. See “How the Diagonals of $A$ are Listed in the Vector $d$” on page 2-3326.

Note that the longest nonzero diagonal in $A$ is contained in column 3 of $B$. The other nonzero diagonals of $A$ have extra zeros added to their corresponding columns in $B$, to give all columns of $B$ the same length. For the nonzero diagonals below the main diagonal of $A$, extra zeros are added at the tops of columns. For the nonzero diagonals above the main diagonal of $A$, extra zeros are added at the bottoms of columns. This is illustrated by the following diagram.
**Example 2**

This example generates a sparse tridiagonal representation of the classic second difference operator on $n$ points.

```matlab
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson's test matrix (see gallery):

```matlab
A = spdiags(abs(-(n-1)/2:(n-1)/2)',0,A)
```

Finally, recover the three diagonals:

```matlab
B = spdiags(A)
```

**Example 3**

The second example is not square.

```matlab
A = [11 0 13 0
     0 22 0 24]
```
Here $m = 7$, $n = 4$, and $p = 3$.

The statement $[B,d] = \text{spdiags}(A)$ produces $d = [-3 \ 0 \ 2]'$ and $B = [41 \ 11 \ 0 \ 52 \ 22 \ 0 \ 63 \ 33 \ 13 \ 74 \ 44 \ 24]$.

Conversely, with the above $B$ and $d$, the expression $\text{spdiags}(B,d,7,4)$ reproduces the original $A$.

**Example 4**

This example shows how $\text{spdiags}$ creates the diagonals when the columns of $B$ are longer than the diagonals they are replacing.

```matlab
B = repmat((1:6)',[1 7])
```

B =

```text
1 1 1 1 1 1 1
2 2 2 2 2 2 2
3 3 3 3 3 3 3
4 4 4 4 4 4 4
5 5 5 5 5 5 5
6 6 6 6 6 6 6
```

```matlab
d = [-4 -2 -1 0 3 4 5];
A = spdiags(B,d,6,6);
full(A)
```

ans =

```text
0 0 33 0
41 0 0 44
0 52 0 0
0 0 63 0
0 0 0 0 74
```
Example 5A

This example illustrates the use of the syntax \( A = \text{spdiags}(B,d,m,n) \), under three conditions:

- \( m \) is equal to \( n \)
- \( m \) is greater than \( n \)
- \( m \) is less than \( n \)

The command used in this example is

\[
A = \text{full}(\text{spdiags}(B, [-2 0 2], m, n))
\]

where \( B \) is the 5-by-3 matrix shown below. The resulting matrix \( A \) has dimensions \( m \)-by-\( n \), and has nonzero diagonals at \([-2 0 2]\) (a sub-diagonal at -2, the main diagonal, and a super-diagonal at 2).

\[
B =
\begin{bmatrix}
1 & 6 & 11 \\
2 & 7 & 12 \\
3 & 8 & 13 \\
4 & 9 & 14 \\
5 & 10 & 15
\end{bmatrix}
\]

The first and third columns of matrix \( B \) are used to create the sub- and super-diagonals of \( A \) respectively. In all three cases though, these two outer columns of \( B \) are longer than the resulting diagonals of \( A \). Because of this, only a part of the columns is used in \( A \).
When \( m = n \) or \( m > n \), \texttt{spdiags} takes elements of the super-diagonal in \( A \) from the lower part of the corresponding column of \( B \), and elements of the sub-diagonal in \( A \) from the upper part of the corresponding column of \( B \).

When \( m < n \), \texttt{spdiags} does the opposite, taking elements of the super-diagonal in \( A \) from the upper part of the corresponding column of \( B \), and elements of the sub-diagonal in \( A \) from the lower part of the corresponding column of \( B \).

**Part 1 — \( m \) is equal to \( n \).**

\[
A = \text{full}(\text{spdiags}(B, [-2 0 2], 5, 5))
\]

Matrix \( B \) | Matrix \( A \)
---|---
1 6 11 | 6 0 13 0 0
2 7 12 | 0 7 0 14 0
3 8 13 | 1 0 8 0 15
4 9 14 | 0 2 0 9 0
5 10 15 | 0 0 3 0 10

\( A(3,1), A(4,2), \) and \( A(5,3) \) are taken from the upper part of \( B(:,1) \).
\( A(1,3), A(2,4), \) and \( A(3,5) \) are taken from the lower part of \( B(:,3) \).

**Part 2 — \( m \) is greater than \( n \).**

\[
A = \text{full}(\text{spdiags}(B, [-2 0 2], 5, 4))
\]

Matrix \( B \) | Matrix \( A \)
---|---
1 6 11 | 6 0 13 0
2 7 12 | 0 7 0 14
3 8 13 | 1 0 8 0
4 9 14 | 0 2 0 9
5 10 15 | 0 0 3 0

Same as in Part A.
Part 3 — m is less than n.

\[
A = \text{full(spdiags(B, [-2 0 2], 4, 5))}
\]

Matrix B        Matrix A
\[
\begin{bmatrix}
1 & 6 & 11 \\
2 & 7 & 12 \\
3 & 8 & 13 \\
4 & 9 & 14 \\
5 & 10 & 15
\end{bmatrix} \rightarrow \begin{bmatrix}
6 & 0 & 11 & 0 & 0 \\
0 & 7 & 0 & 12 & 0 \\
3 & 0 & 8 & 0 & 13 \\
0 & 4 & 0 & 9 & 0 \\
0 & 0 & 3 & 0 & 10
\end{bmatrix}
\]

A(3,1) and A(4,2) are taken from the lower part of B(:,1).
A(1,3), A(2,4), and A(3,5) are taken from the upper part of B(:,3).

Example 5B

Extract the diagonals from the first part of this example back into a column format using the command

\[
B = \text{spdiags}(A)
\]

You can see that in each case the original columns are restored (minus those elements that had overflowed the super- and sub-diagonals of matrix A).

Part 1.

<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Matrix B</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 0 13 0 0</td>
<td>1 6 0</td>
</tr>
<tr>
<td>0 7 0 14 0</td>
<td>2 7 0</td>
</tr>
<tr>
<td>1 0 8 0 15</td>
<td>3 8 13</td>
</tr>
<tr>
<td>0 2 0 9 0</td>
<td>0 9 14</td>
</tr>
<tr>
<td>0 0 3 0 10</td>
<td>0 10 15</td>
</tr>
</tbody>
</table>

Part 2.

<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Matrix B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-3333
spdiags

\[
\begin{bmatrix}
6 & 0 & 13 & 0 \\
0 & 7 & 0 & 14 \\
1 & 0 & 8 & 0 \\
0 & 2 & 0 & 9 \\
0 & 0 & 3 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 6 & 0 \\
2 & 7 & 0 \\
3 & 8 & 13 \\
0 & 9 & 14 \\
0 & 0 & 0
\end{bmatrix}
\]

**Part 3.**

<table>
<thead>
<tr>
<th>Matrix A</th>
<th>Matrix B</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 0 11 0 0</td>
<td>0 6 11</td>
</tr>
<tr>
<td>0 7 0 12 0</td>
<td>0 7 12</td>
</tr>
<tr>
<td>3 0 8 0 13</td>
<td>3 8 13</td>
</tr>
<tr>
<td>0 4 0 9 0</td>
<td>4 9 0</td>
</tr>
</tbody>
</table>

**See Also**

diag, speye
**Purpose**
Calculate specular reflectance

**Syntax**
\[ R = \text{specular}(Nx,Ny,Nz,S,V) \]

**Description**
\[ R = \text{specular}(Nx,Ny,Nz,S,V) \] returns the reflectance of a surface with normal vector components \([Nx,Ny,Nz]\). \(S\) and \(V\) specify the direction to the light source and to the viewer, respectively. You can specify these directions as three vectors \([x,y,z]\) or two vectors \([\Theta \Phi]\) (in spherical coordinates).

The specular highlight is strongest when the normal vector is in the direction of \((S+V)/2\) where \(S\) is the source direction, and \(V\) is the view direction.

The surface spread exponent can be specified by including a sixth argument as in \(\text{specular}(Nx,Ny,Nz,S,V,\text{spread})\).
### Purpose
Sparse identity matrix

### Syntax
- `S = speye(m,n)`
- `S = speye(n)`

### Description
- `S = speye(m,n)` forms an `m`-by-`n` sparse matrix with 1s on the main diagonal.
- `S = speye(n)` abbreviates `speye(n,n)`.

### Examples
- `I = speye(1000)` forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as `I = sparse(eye(1000,1000))`, but the latter requires eight megabytes for temporary storage for the full representation.

### See Also
- `spalloc`, `spones`, `spdiags`, `sprand`, `sprandn`
**Purpose**
Apply function to nonzero sparse matrix elements

**Syntax**
f = spfun(fun,S)

**Description**
The `spfun` function selectively applies a function to only the nonzero elements of a sparse matrix `S`, preserving the sparsity pattern of the original matrix (except for underflow or if `fun` returns zero for some nonzero elements of `S`).

`f = spfun(fun,S)` evaluates `fun(S)` on the nonzero elements of `S`. `fun` is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions” in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `fun`, if necessary.

**Remarks**
Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

**Examples**
Given the 4-by-4 sparse diagonal matrix

\[ S = spdiags([1:4]',0,4,4) \]

\[
S = \\
(1,1)\quad 1 \\
(2,2)\quad 2 \\
(3,3)\quad 3 \\
(4,4)\quad 4 
\]

Because `fun` returns nonzero values for all nonzero element of `S`, `f = spfun(@exp,S)` has the same sparsity pattern as `S`.

\[
f = \\
(1,1)\quad 2.7183 \\
(2,2)\quad 7.3891 \\
(3,3)\quad 20.0855 \\
(4,4)\quad 54.5982 
\]
whereas $\exp(S)$ has 1s where $S$ has 0s.

```matlab
full(exp(S))
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7183</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>7.3891</td>
<td>1.0000</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>20.0855</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>54.5982</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

function_handle (@)
**Purpose**
Transform spherical coordinates to Cartesian

**Syntax**

\[ [x, y, z] = \text{sph2cart}(\text{THETA}, \text{PHI}, \text{R}) \]

**Description**

\[ [x, y, z] = \text{sph2cart}(\text{THETA}, \text{PHI}, \text{R}) \]
transforms the corresponding elements of spherical coordinate arrays to Cartesian, or \( xyz \), coordinates. THETA, PHI, and R must all be the same size (or any of them can be scalar). THETA and PHI are angular displacements in radians from the positive \( x \)-axis and from the \( x-y \) plane, respectively.

**Algorithm**
The mapping from spherical coordinates to three-dimensional Cartesian coordinates is

\[
\begin{align*}
x &= r \cdot \cos(\phi) \cdot \cos(\theta) \\
y &= r \cdot \cos(\phi) \cdot \sin(\theta) \\
z &= r \cdot \sin(\phi)
\end{align*}
\]

**See Also**
cart2pol, cart2sph, pol2cart
**Purpose**
Generate sphere

![sphere](image)

**Syntax**
sphere
sphere(n)
[X,Y,Z] = sphere(n)

**Description**
The `sphere` function generates the x-, y-, and z-coordinates of a unit sphere for use with `surf` and `mesh`.

- `sphere` generates a sphere consisting of 20-by-20 faces.
- `sphere(n)` draws a `surf` plot of an n-by-n sphere in the current figure.
- `[X,Y,Z] = sphere(n)` returns the coordinates of a sphere in three matrices that are (n+1)-by-(n+1) in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

**Examples**
Generate and plot a sphere.

```matlab
sphere
axis equal
```
See Also
cylinder, axis equal
“Polygons and Surfaces” on page 1-97 for related functions
Purpose
Spin colormap

Syntax
spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')

Description
The spinmap function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

spinmap cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

spinmap(t) rotates the colormap for approximately $10^t$ seconds. The amount of time specified by $t$ depends on your hardware configuration (e.g., if you are running MATLAB software over a network).

spinmap(t,inc) rotates the colormap for approximately $10^t$ seconds and specifies an increment $inc$ by which the colormap shifts. When $inc$ is 1, the rotation appears smoother than the default (i.e., 2). Increments greater than 2 are less smooth than the default. A negative increment (e.g., −2) rotates the colormap in a negative direction.

spinmap('inf') rotates the colormap for an infinite amount of time. To break the loop, press Ctrl+C.

See Also
colormap, colormapeditor
“Color Operations” on page 1-105 for related functions
Purpose

Cubic spline data interpolation

Syntax

\[
\begin{align*}
    yy &= \text{spline}(x,Y,xx) \\
    pp &= \text{spline}(x,Y)
\end{align*}
\]

Description

\(yy = \text{spline}(x,Y,xx)\) uses a cubic spline interpolation to find \(yy\), the values of the underlying function \(Y\) at the values of the interpolant \(xx\). For the interpolation, the independent variable is assumed to be the final dimension of \(Y\) with the breakpoints defined by \(x\).

The sizes of \(xx\) and \(yy\) are related as follows:

- If \(Y\) is a scalar or vector, \(yy\) has the same size as \(xx\).
- If \(Y\) is an array that is not a vector,
  - If \(xx\) is a scalar or vector, \text{size}(yy) equals \([d_1, d_2, \ldots, d_k, \text{length}(xx)]\).
  - If \(xx\) is an array of size \([m_1,m_2,\ldots,m_j]\), \text{size}(yy) equals \([d_1,d_2,\ldots,d_k,m_1,m_2,\ldots,m_j]\).

\(pp = \text{spline}(x,Y)\) returns the piecewise polynomial form of the cubic spline interpolant for later use with \text{ppval} and the spline utility \text{unmkpp}.

\(x\) must be a vector. \(Y\) can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If \(Y\) is a scalar or vector, it must have the same length as \(x\). A scalar value for \(x\) or \(Y\) is expanded to have the same length as the other. See Exceptions (1) for an exception to this rule, in which the not-a-knot end conditions are used.
- If \(Y\) is an array that is not a vector, the size of \(Y\) must have the form \([d_1,d_2,\ldots,d_k,n]\), where \(n\) is the length of \(x\). The interpolation is performed for each \(d_1\)-by-\(d_2\)-by-\(\ldots\)-\(d_k\) value in \(Y\). See Exceptions (2) for an exception to this rule.
Exceptions

1 If \( Y \) is a vector that contains two more values than \( x \) has entries, the first and last value in \( Y \) are used as the endslopes for the cubic spline. If \( Y \) is a vector, this means
   - \( f(x) = Y(2:end-1) \)
   - \( df(\text{min}(x)) = Y(1) \)
   - \( df(\text{max}(x)) = Y(\text{end}) \)

2 If \( Y \) is a matrix or an \( N \)-dimensional array with \( \text{size}(Y,N) \) equal to \( \text{length}(x)+2 \), the following hold:
   - \( f(x(j)) \) matches the value \( Y(:,:,...,j+1) \) for \( j=1:\text{length}(x) \)
   - \( Df(\text{min}(x)) \) matches \( Y(:,:,...,1) \)
   - \( Df(\text{max}(x)) \) matches \( Y(:,:,...,\text{end}) \)

Note You can also perform spline interpolation using the \texttt{interp1} function with the command \texttt{interp1(x,y,xx,'spline')}. Note that while \texttt{spline} performs interpolation on rows of an input matrix, \texttt{interp1} performs interpolation on columns of an input matrix.

Examples

Example 1

This generates a sine curve, then samples the spline over a finer mesh.

\[
x = 0:10;
y = \sin(x);
xx = 0:.25:10;
yy = \text{spline}(x,y,xx);
\text{plot}(x,y,'o',xx,yy)
\]
Example 2

This illustrates the use of clamped or complete spline interpolation where end slopes are prescribed. Zero slopes at the ends of an interpolant to the values of a certain distribution are enforced.

\[
\begin{align*}
x &= -4:4; \\
y &= [0 \ 0.15 \ 1.12 \ 2.36 \ 2.36 \ 1.46 \ 0.49 \ 0.06 \ 0]; \\
cs &= \text{spline}(x,[0 \ y \ 0]); \\
xx &= \text{linspace}(-4,4,101); \\
\text{plot}(x,y,'o',xx,\text{ppval}(cs,xx),'-');
\end{align*}
\]
Example 3
The two vectors

\[
\begin{align*}
t &= 1900:10:1990; \\
p &= [ 75.995 \quad 91.972 \quad 105.711 \quad 123.203 \quad 131.669 \ldots \\
&\quad \quad \quad 150.697 \quad 179.323 \quad 203.212 \quad 226.505 \quad 249.633 ];
\end{align*}
\]

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

\[
spline(t,p,2000)
\]

uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

\[
\text{ans} = \\
270.6060
\]
Example 4

The statements

```matlab
x = pi*[0:.5:2];
y = [0 1 0 -1 0 1 0; 1 0 1 0 -1 0 1];
pp = spline(x,y);
yy = ppval(pp, linspace(0,2*pi,101));
plot(yy(:,1),yy(:,2),'-b',y(1,2:5),y(2,2:5),'or'), axis equal
```

generate the plot of a circle, with the five data points \(y(:,2), \ldots, y(:,6)\) marked with o's. Note that this \(y\) contains two more values (i.e., two more columns) than does \(x\), hence \(y(:,1)\) and \(y(:,\text{end})\) are used as endslopes.
Example 5

The following code generates sine and cosine curves, then samples the splines over a finer mesh.

```matlab
x = 0:.25:1;
Y = [sin(x); cos(x)];
xx = 0:.1:1;
YY = spline(x,Y,xx);
plot(x,Y(1,:),'o',xx,YY(1,:),'-'); hold on;
plot(x,Y(2,:),'o',xx,YY(2,:),':'); hold off;
```

Algorithm

A tridiagonal linear system (with, possibly, several right sides) is being solved for the information needed to describe the coefficients of the various cubic polynomials which make up the interpolating spline. `spline` uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines
form a small suite of functions for working with piecewise polynomials. For access to more advanced features, see the M-file help for these functions and the Spline Toolbox.

See Also

interp1, ppval, mkpp, pchip, unmkpp

References

**spones**

**Purpose**
Replace nonzero sparse matrix elements with ones

**Syntax**
R = spones(S)

**Description**
R = spones(S) generates a matrix R with the same sparsity structure as S, but with 1’s in the nonzero positions.

**Examples**
c = sum(spones(S)) is the number of nonzeros in each column.
r = sum(spones(S'))' is the number of nonzeros in each row.
sum(c) and sum(r) are equal, and are equal to nnz(S).

**See Also**
nnz, spalloc, spfun
**Purpose**

Set parameters for sparse matrix routines

**Syntax**

```matlab
spparms('key',value)
spparms
values = spparms
[ keys,values ] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

**Description**

`spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse routines, particularly the minimum degree orderings, `colmmd` and `symmmd`, and also within sparse backslash. In ordinary use, you should never need to deal with this function.

The meanings of the `key` parameters are

- `'spumoni'`  
  Sparse Monitor flag:
  - 0  Produces no diagnostic output, the default
  - 1  Produces information about choice of algorithm based on matrix structure, and about storage allocation
  - 2  Also produces very detailed information about the sparse matrix algorithms

- `'thr_rel'`, `'thr_abs'`  
  Minimum degree threshold is `thr_rel*mindegree + thr_abs`.

- `'exact_d'`  
  Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.

- `'supernd'`  
  If positive, minimum degree amalgamates the supernodes every `supernd` stages.
spparms

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'rreduce'</td>
<td>If positive, minimum degree does row reduction every rreduce stages.</td>
</tr>
<tr>
<td>'wh_frac'</td>
<td>Rows with density &gt; wh_frac are ignored in colmmd.</td>
</tr>
<tr>
<td>'autommd'</td>
<td>Nonzero to use minimum degree (MMD) orderings with QR-based \ and /.</td>
</tr>
<tr>
<td>'autoamd'</td>
<td>Nonzero to use colamd ordering with the UMFPACK LU-based \ and /, and to use amd with CHOLMOD Cholesky-based \ and /.</td>
</tr>
<tr>
<td>'piv_tol'</td>
<td>Pivot tolerance used by the UMFPACK LU-based \ and /.</td>
</tr>
<tr>
<td>'bandden'</td>
<td>Band density used by LAPACK-based \ and / for banded matrices. Band density is defined as (# nonzeros in the band)/(# nonzeros in a full band). If bandden = 1.0, never use band solver. If bandden = 0.0, always use band solver. Default is 0.5.</td>
</tr>
<tr>
<td>'umfpack'</td>
<td>Nonzero to use UMFPACK instead of the v4 LU-based solver in \ and /.</td>
</tr>
<tr>
<td>'sym_tol'</td>
<td>Symmetric pivot tolerance used by UMFPACK. See lu for more information about the role of the symmetric pivot tolerance.</td>
</tr>
</tbody>
</table>

**Note** LU-based \ and / (UMFPACK) on square matrices use a modified colamd or amd. Cholesky-based \ and / (CHOLMOD) on symmetric positive definite matrices use amd. QR-based \ and / on rectangular matrices use colmmd.

spparms, by itself, prints a description of the current settings.

values = spparms returns a vector whose components give the current settings.
[keys, values] = spparms returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

spparms(values), with no output argument, sets all the parameters to the values specified by the argument vector.

value = spparms('key') returns the current setting of one parameter.

spparms('default') sets all the parameters to their default settings.

spparms('tight') sets the minimum degree ordering parameters to their tight settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for default and tight settings are

<table>
<thead>
<tr>
<th>values(1)</th>
<th>'spumoni'</th>
<th>Default</th>
<th>Tight</th>
</tr>
</thead>
<tbody>
<tr>
<td>values(2)</td>
<td>'thr_rel'</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>values(3)</td>
<td>'thr_abs'</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>values(4)</td>
<td>'exact_d'</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>values(5)</td>
<td>'supernd'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>values(6)</td>
<td>'rreduce'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>values(7)</td>
<td>'wh_frac'</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>values(8)</td>
<td>'autommd'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>values(9)</td>
<td>'autoamd'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>values(10)</td>
<td>'piv_tol'</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>values(11)</td>
<td>'bandden'</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>values(12)</td>
<td>'umfpack'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>values(13)</td>
<td>'sym_tol'</td>
<td>0.001</td>
<td></td>
</tr>
</tbody>
</table>
### Notes

**Sparse A\b on Symmetric Positive Definite A**

Sparse A\b on symmetric positive definite A uses CHOLMOD in conjunction with the amd reordering routine.

The parameter 'autoamd' turns the amd reordering on or off within the solver.

**Sparse A\b on General Square A**

Sparse A\b on general square A usually uses UMFPACK in conjunction with amd or a modified colamd reordering routine.

The parameter 'umfpack' turns the use of the UMFPACK software on or off within the solver.

If UMFPACK is used,

- The parameter 'piv_tol' controls pivoting within the solver.
- The parameter 'autoamd' turns amd and the modified colamd on or off within the solver.

If UMFPACK is not used,

- An LU-based solver is used in conjunction with the colmmd reordering routine.
- If UMFPACK is not used, then the parameter 'autommd' turns the colmmd reordering routine on or off within the solver.
- If UMFPACK is not used and colmmd is used within the solver, then the minimum degree parameters affect the reordering routine within the solver.

**Sparse A\b on Rectangular A**

Sparse A\b on rectangular A uses a QR-based solve in conjunction with the colmmd reordering routine.

The parameter 'autommd' turns the colmmd reordering on or off within the solver.
If `colmmd` is used within the solver, then the minimum degree parameters affect the reordering routine within the solver.

See Also

\, chol, lu, qr, colamd symamd

References


**Purpose**  
Sparse uniformly distributed random matrix

**Syntax**
- `R = sprand(S)`  
- `R = sprand(m,n,density)`  
- `R = sprand(m,n,density,rc)`

**Description**
- `R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

- `R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (0 <= density <= 1).

- `R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

`sprand` uses the internal state information set with the `rand` function.

**See Also**
- `sprandn`, `sprandsym`
Purpose
Sparse normally distributed random matrix

Syntax
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)

Description
R = sprandn(S) has the same sparsity structure as S, but normally distributed random entries with mean 0 and variance 1.

R = sprandn(m,n,density) is a random, m-by-n, sparse matrix with approximately density*m*n normally distributed nonzero entries (0 <= density <= 1).

R = sprandn(m,n,density,rc) also has reciprocal condition number approximately equal to rc. R is constructed from a sum of matrices of rank one.

If rc is a vector of length lr, where lr <= min(m,n), then R has rc as its first lr singular values, all others are zero. In this case, R is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

sprandn uses the internal state information set with the randn function.

See Also
sprand, sprandsym
**Purpose**
Sparse symmetric random matrix

**Syntax**
- `R = sprandsym(S)`
- `R = sprandsym(n,density)`
- `R = sprandsym(n,density,rc)`
- `R = sprandsym(n,density,rc,kind)`

**Description**
- `R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as S. Its elements are normally distributed, with mean 0 and variance 1.

- `R = sprandsym(n,density)` returns a symmetric random, n-by-n, sparse matrix with approximately density*n*n nonzeros; each entry is the sum of one or more normally distributed random samples, and (0 <= density <= 1).

- `R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to rc. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in [-1, 1].

  If rc is a vector of length n, then R has eigenvalues rc. Thus, if rc is a positive (nonnegative) vector then R is a positive definite matrix. In either case, R is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

- `R = sprandsym(n,density,rc,kind)` returns a positive definite matrix. Argument kind can be:
  - 1 to generate R by random Jacobi rotation of a positive definite diagonal matrix. R has the desired condition number exactly.
  - 2 to generate an R that is a shifted sum of outer products. R has the desired condition number only approximately, but has less structure.
  - 3 to generate an R that has the same structure as the matrix S and approximate condition number 1/rc. density is ignored.

**See Also**
sprand, sprandn
Purpose       Structural rank

Syntax        r = sprank(A)

Description   r = sprank(A) is the structural rank of the sparse matrix A. For all
values of A,

    sprank(A) >= rank(full(A))

In exact arithmetic, sprank(A) == rank(full(sprandn(A))) with
a probability of one.

Examples       A = [ 1  0  2  0
                      2  0  4  0 ];

                A = sparse(A);

                sprank(A)

                ans =
                      2

                rank(full(A))

                ans =
                      1

See Also       dmperm
**Purpose**
Write formatted data to string

**Syntax**

```
[s, errmsg] = sprintf(format, A, ...)
```

**Description**

```
[s, errmsg] = sprintf(format, A, ...) formats the data in matrix A (and in any additional matrix arguments) under control of the specified format string and returns it in the MATLAB string variable s. The sprintf function returns an error message string errmsg if an error occurred. errmsg is an empty matrix if no error occurred.
```
sprintf is the same as fprintf except that it returns the data in a MATLAB string variable rather than writing it to a file.

See “Formatting Strings” in the MATLAB Programming Fundamentals documentation for more detailed information on using string formatting commands.

**Format String**

The format argument is a string containing ordinary characters and/or C language conversion specifications. A conversion specification controls the notation, alignment, significant digits, field width, and other aspects of output format. The format string can contain escape characters to represent nonprinting characters such as newline characters and tabs.

Conversion specifications begin with the % character and contain these optional and required elements:

- Flags (optional)
- Width and precision fields (optional)
- A subtype specifier (optional)
- Conversion character (required)

You specify these elements in the following order:
**Flags**

You can control the alignment of the output using any of these optional flags.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A minus sign (-)</td>
<td>Left-justifies the converted argument in its field</td>
<td>% 5.2d</td>
</tr>
<tr>
<td>A plus sign (+)</td>
<td>Always prints a sign character (+ or –)</td>
<td>%+5.2d</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Pad with zeros rather than spaces.</td>
<td>%05.2f</td>
</tr>
</tbody>
</table>

**Field Width and Precision Specifications**

You can control the width and precision of the output by including these options in the format string.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field width</td>
<td>A string specifying the minimum number of characters to be printed. This includes a plus or minus sign, any leading zeros, numeric digits, and decimal point.</td>
<td>%6f</td>
</tr>
<tr>
<td>Precision</td>
<td>A string including a period (.) specifying the number of digits to be printed to the right of the decimal point</td>
<td>%6.2f</td>
</tr>
</tbody>
</table>
Conversion Characters

Conversion characters specify the notation of the output.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>Single character</td>
</tr>
<tr>
<td>%d</td>
<td>Decimal notation (signed)</td>
</tr>
<tr>
<td>%e</td>
<td>Exponential notation (using a lowercase e as in 3.1415e+00)</td>
</tr>
<tr>
<td>%E</td>
<td>Exponential notation (using an uppercase E as in 3.1415E+00)</td>
</tr>
<tr>
<td>%f</td>
<td>Fixed-point notation</td>
</tr>
<tr>
<td>%g</td>
<td>The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.</td>
</tr>
<tr>
<td>%G</td>
<td>Same as %g, but using an uppercase E</td>
</tr>
<tr>
<td>%o</td>
<td>Octal notation (unsigned)</td>
</tr>
<tr>
<td>%s</td>
<td>String of characters</td>
</tr>
<tr>
<td>%u</td>
<td>Decimal notation (unsigned)</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal notation (using lowercase letters a–f)</td>
</tr>
<tr>
<td>%X</td>
<td>Hexadecimal notation (using uppercase letters A–F)</td>
</tr>
</tbody>
</table>

The following tables describe the nonalphanumeric characters found in format specification strings.

Escape Characters

This table lists the escape character sequences you use to specify non-printing characters in a format specification.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
</tbody>
</table>
## Remarks

The `sprintf` function behaves like its ANSI C language namesake with these exceptions and extensions.

- If you use `sprintf` to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the `fix`, `floor`, `ceil`, or `round` functions to change the value in the double into a value that can be represented as an integer before passing it to `sprintf`.

- The following nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>New line</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>''(two single quotes)</td>
<td>Single quotation mark</td>
</tr>
<tr>
<td>%%</td>
<td>Percent character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '%bx'.</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>The underlying C data type is a float rather than an unsigned integer. For example, to print a double value in hexadecimal use the format '%bx'.</td>
</tr>
</tbody>
</table>

- The `sprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of A
(columnwise) until all the elements are used up. The function then continues in a similar manner through any additional matrix arguments.

- If %s is used to print part of a nonscalar double argument, the following behavior occurs:

1. Successive values are printed as long as they are integers and in the range of a valid character. The first invalid character terminates the printing for this %s specifier and is used for a later specifier. For example, pi terminates the string below and is printed using %f format.

   ```
   Str = [65 66 67 pi];
   sprintf('%s %f', Str)
   ans =
   ABC 3.141593
   ```

2. If the first value to print is not a valid character, then just that value is printed for this %s specifier using an e conversion as a warning to the user. For example, pi is formatted by %s below in exponential notation, and 65, though representing a valid character, is formatted as fixed-point (%f).

   ```
   Str = [pi 65 66 67];
   sprintf('%s %f %s', Str)
   ans =
   3.141593e+000 65.000000 BC
   ```

3. One exception is zero, which is a valid character. If zero is found first, %s prints nothing and the value is skipped. If zero is found after at least one valid character, it terminates the printing for this %s specifier and is used for a later specifier.

- `sprintf` prints exponents differently on some platforms, as shown in the following table:
Exponents Printed with %e, %E, %g, or %G

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum Digits in Exponent</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>3</td>
<td>1.23e+004</td>
</tr>
<tr>
<td>UNIX</td>
<td>2</td>
<td>1.23e+04</td>
</tr>
</tbody>
</table>

You can resolve this difference in exponents by postprocessing the results of sprintf. For example, to make the PC output look like that of UNIX, use

```matlab
a = sprintf('%e', 12345.678);
if ispc, a = strrep(a, 'e+0', 'e+'); end
```

### Examples

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>sprintf('%0.5g',(1+sqrt(5))/2)</td>
<td>1.618</td>
</tr>
<tr>
<td>sprintf('%0.5g',1/eps)</td>
<td>4.5036e+15</td>
</tr>
<tr>
<td>sprintf('%15.5f',1/eps)</td>
<td>4503599627370496.00000</td>
</tr>
<tr>
<td>sprintf('%d',round(pi))</td>
<td>3</td>
</tr>
<tr>
<td>sprintf('%s','hello')</td>
<td>hello</td>
</tr>
<tr>
<td>sprintf('The array is %dx%d.','2,3')</td>
<td>The array is 2x3</td>
</tr>
<tr>
<td>sprintf('\n')</td>
<td>Line termination character on all platforms</td>
</tr>
</tbody>
</table>

### See Also

int2str, num2str, scanf

### References

Purpose

Visualize sparsity pattern

Syntax

spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',markersize)

Description

plots the

spy(S) sparsity pattern of any matrix S.

spy(S,markersize), where markersize is an integer, plots the sparsity pattern using markers of the specified point size.

spy(S,'LineSpec'), where LineSpec is a string, uses the specified plot marker type and color.

spy(S,'LineSpec',markersize) uses the specified type, color, and size for the plot markers.

S is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

Note spy replaces format +, which takes much more space to display essentially the same information.

Examples

This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

B = bucky;
spy(B)
See Also
find, gplot, LineSpec, symamd, symrcm
Purpose

Square root

Syntax

B = sqrt(X)

Description

B = sqrt(X) returns the square root of each element of the array X. For the elements of X that are negative or complex, sqrt(X) produces complex results.

Remarks

See sqrtm for the matrix square root.

Examples

sqrt((-2:2)')
ans =
0 + 1.4142i
0 + 1.0000i
0
1.0000
1.4142

See Also

sqrtm, realsqrt
**Purpose**

Matrix square root

**Syntax**

X = sqrtm(A)

[X, resnorm] = sqrtm(A)

[X, alpha, condest] = sqrtm(A)

**Description**

X = sqrtm(A) is the principal square root of the matrix A, i.e. X*X = A. X is the unique square root for which every eigenvalue has nonnegative real part. If A has any eigenvalues with negative real parts then a complex result is produced. If A is singular then A may not have a square root. A warning is printed if exact singularity is detected.

[X, resnorm] = sqrtm(A) does not print any warning, and returns the residual, \( \frac{\|A - X^2\|_F}{\|A\|_F} \).

[X, alpha, condest] = sqrtm(A) returns a stability factor alpha and an estimate condest of the matrix square root condition number of X. The residual \( \frac{\|A - X^2\|_F}{\|A\|_F} \) is bounded approximately by \( n*alpha*\text{eps} \) and the Frobenius norm relative error in X is bounded approximately by \( n*alpha*condest*\text{eps} \), where \( n = \max(\text{size}(A)) \).

**Remarks**

If X is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like \( X = \begin{bmatrix} 0 & 1; & 0 & 0 \end{bmatrix} \), do not have any square roots, real or complex, and sqrtm cannot be expected to produce one.

**Examples**

**Example 1**

A matrix representation of the fourth difference operator is

\[
X = \begin{bmatrix}
5 & -4 & 1 & 0 & 0 \\
-4 & 6 & -4 & 1 & 0 \\
1 & -4 & 6 & -4 & 1 \\
0 & 1 & -4 & 6 & -4 \\
0 & 0 & 1 & -4 & 5
\end{bmatrix}
\]
This matrix is symmetric and positive definite. Its unique positive definite square root, \( Y = \text{sqrtm}(X) \), is a representation of the second difference operator.

\[
Y =
\begin{pmatrix}
2 & -1 & -0 & -0 & -0 \\
-1 & 2 & -1 & 0 & -0 \\
0 & -1 & 2 & -1 & 0 \\
-0 & 0 & -1 & 2 & -1 \\
-0 & -0 & -0 & 2 & 0 \\
\end{pmatrix}
\]

**Example 2**

The matrix

\[
X =
\begin{pmatrix}
7 & 10 \\
15 & 22 \\
\end{pmatrix}
\]

has four square roots. Two of them are

\[
Y_1 =
\begin{pmatrix}
1.5667 & 1.7408 \\
2.6112 & 4.1779 \\
\end{pmatrix}
\]

and

\[
Y_2 =
\begin{pmatrix}
1 & 2 \\
3 & 4 \\
\end{pmatrix}
\]

The other two are \(-Y_1\) and \(-Y_2\). All four can be obtained from the eigenvalues and vectors of \( X \).

\[
[V,D] = \text{eig}(X);
D =
\begin{pmatrix}
0.1386 & 0 \\
0 & 28.8614 \\
\end{pmatrix}
\]
The four square roots of the diagonal matrix $D$ result from the four choices of sign in

\[
S =
\begin{pmatrix}
-0.3723 & 0 \\
0 & -5.3723
\end{pmatrix}
\]

All four $Y$s are of the form

\[
Y = V*S/V
\]

The `sqrtm` function chooses the two plus signs and produces $Y1$, even though $Y2$ is more natural because its entries are integers.

**See Also**

`expm`, `funm`, `logm`
Purpose

Remove singleton dimensions

Syntax

B = squeeze(A)

Description

B = squeeze(A) returns an array B with the same elements as A, but with all singleton dimensions removed. A singleton dimension is any dimension for which \texttt{size(A,dim)} = 1. Two-dimensional arrays are unaffected by \texttt{squeeze}; if A is a row or column vector or a scalar (1-by-1) value, then \( B = A \).

Examples

Consider the 2-by-1-by-3 array \( Y = \text{rand}(2,1,3) \). This array has a singleton column dimension — that is, there’s only one column per page.

\[
Y = \\
Y(:,:,1) = \\
\begin{bmatrix}
0.5194 & 0.0346 \\
0.8310 & 0.0535
\end{bmatrix} \\
Y(:,:,2) = \\
\begin{bmatrix}
0.5194 & 0.0346 \\
0.8310 & 0.0535
\end{bmatrix} \\
Y(:,:,3) = \\
\begin{bmatrix}
0.5297 \\
0.6711
\end{bmatrix}
\]

The command \( Z = \text{squeeze}(Y) \) yields a 2-by-3 matrix:

\[
Z = \\
\begin{bmatrix}
0.5194 & 0.0346 & 0.5297 \\
0.8310 & 0.0535 & 0.6711
\end{bmatrix}
\]

Consider the 1-by-1-by-5 array \( \text{mat} = \text{repmat}(1,[1,1,5]) \). This array has only one scalar value per page.

\[
\text{mat} = \\
\text{mat}(:,:,1) = \text{mat}(:,:,2) = \\
1 & 1
\]
mat(:,:,3) = mat(:,:,4) =
    1     1
mat(:,:,5) =
    1

The command squeeze(mat) yields a 5-by-1 matrix:
squeeze(mat)
ans =
    1
    1
    1
    1
    1
    1

size(squeeze(mat))
ans =
    5     1

See Also reshape, shiftdim
Purpose
Convert state-space filter parameters to transfer function form

Syntax
[b, a] = ss2tf(A, B, C, D, iu)

Description
ss2tf converts a state-space representation of a given system to an equivalent transfer function representation.

[b, a] = ss2tf(A, B, C, D, iu) returns the transfer function

\[ H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D \]

of the system

\[ \dot{x} = Ax + Bu \]
\[ y = Cx + Du \]

from the iu-th input. Vector a contains the coefficients of the denominator in descending powers of s. The numerator coefficients are returned in array b with as many rows as there are outputs y. ss2tf also works with systems in discrete time, in which case it returns the z-transform representation.

The ss2tf function is part of the standard MATLAB language.

Algorithm
The ss2tf function uses poly to find the characteristic polynomial \( \det(sI-A) \) and the equality:

\[ H(s) = C(sI - A)^{-1}B = \frac{\det(sI - A + BC) - \det(sI - A)}{\det(sI - A)} \]
**Purpose**
Read formatted data from string

**Syntax**

\[ \text{A} = \text{sscanf}(\text{s}, \text{format}) \]
\[ \text{A} = \text{sscanf}(\text{s}, \text{format}, \text{size}) \]
\[ [\text{A}, \text{count}, \text{errmsg}, \text{nextindex}] = \text{sscanf}(\ldots) \]

**Description**

\[ \text{A} = \text{sscanf}(\text{s}, \text{format}) \]
reads data from the MATLAB string \( \text{s} \), converts it according to the specified format string, and returns it in matrix \( \text{A} \). format is a string specifying the format of the data to be read. See "Remarks" for details. sscanf is the same as fscanf except that it reads the data from a MATLAB string rather than reading it from a file. If \( \text{s} \) is a character array with more than one row, sscanf reads the characters in column order.

\[ \text{A} = \text{sscanf}(\text{s}, \text{format}, \text{size}) \]
reads the amount of data specified by size and converts it according to the specified format string. size is an argument that determines how much data is read. Valid options are

<table>
<thead>
<tr>
<th>n</th>
<th>Read at most n numbers, characters, or strings.</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf</td>
<td>Read to the end of the input string.</td>
</tr>
<tr>
<td>[m,n]</td>
<td>Read at most (m*n) numbers, characters, or strings. Fill a matrix of at most m rows in column order. n can be inf, but m cannot.</td>
</tr>
</tbody>
</table>

Characteristics of the output matrix \( \text{A} \) depend on the values read from the input string and on the size argument. If \( \text{sscanf} \) reads only numbers, and if size is not of the form \([m,n]\), matrix \( \text{A} \) is a column vector of numbers. If \( \text{sscanf} \) reads only characters or strings, and if size is not of the form \([m,n]\), matrix \( \text{A} \) is a row vector of characters. See the Remarks section for more information.

\( \text{sscanf} \) differs from its C language namesake scanf() in an important respect — it is vectorized to return a matrix argument. The format string is cycled through the input string until the first of these conditions occurs:

- The format string fails to match the data in the input string
• The amount of data specified by \texttt{size} is read

• The end of the string is reached

\[ [A, \text{count}, \text{errmsg}, \text{nextindex}] = \text{sscanf}(...) \] reads data from the MATLAB string (character array) \texttt{s}, converts it according to the specified \texttt{format} string, and returns it in matrix \texttt{A}. \texttt{count} is an optional output argument that returns the number of values successfully read. \texttt{errmsg} is an optional output argument that returns an error message string if an error occurred or an empty string if an error did not occur. \texttt{nextindex} is an optional output argument specifying one more than the number of characters scanned in \texttt{s}.

\textbf{Remarks}

When MATLAB reads a specified string, it attempts to match the data in the input string to the format string. If a match occurs, the data is written into the output matrix. If a partial match occurs, only the matching data is written to the matrix, and the read operation stops.

The \texttt{format} string consists of ordinary characters and/or conversion specifications. Conversion specifications indicate the type of data to be matched and involve the character \%\%, optional width fields, and conversion characters, organized as shown below:

\begin{center}
\begin{tikzpicture}
\node [align=center] (initial) at (0,0) {Initial \% character};
\node (flag) at (1,0) {Flag};
\node (field) at (2,0) {Field width};
\node (conv) at (3,0) {Conversion character};
\draw [->] (initial) -- node [pos=0.5] {\%12e} (flag);
\draw [->] (flag) -- (field);
\draw [->] (field) -- (conv);
\end{tikzpicture}
\end{center}

Add one or more of these characters between the \% and the conversion character.
An asterisk (*) | Skip over the matched value and do not store it in the output matrix
---|---
A digit string | Maximum field width
A letter | The size of the receiving object; for example, h for short, as in %hd for a short integer, or l for long, as in %ld for a long integer or %lg for a double floating-point number

Valid conversion characters are as shown.

| %c | Sequence of characters; number specified by field width |
| %d | Base 10 integers |
| %e, %f, %g | Floating-point numbers |
| %i | Defaults to signed base 10 integers. Data starting with 0 is read as base 8. Data starting with 0x or 0X is read as base 16. |
| %o | Signed octal integer returned as unsigned |
| %s | A series of non-white-space characters |
| %u | Unsigned decimal |
| %x | Signed hexadecimal integer returned as unsigned |
| [... | Sequence of characters (scanlist) |

Format specifiers %e, %f, and %g accept the text 'inf', '-inf', 'nan', and '-nan'. This text is not case sensitive. The sscanf function converts these to the numeric representation of Inf, -Inf, NaN, and -NaN.

Use %c to read space characters, or %s to skip all white space.

For more information about format strings, refer to the sscanf() and fscanf() routines in a C language reference manual.
Output Characteristics: Only Numeric Values Read

Format characters that cause `sscanf` to read numbers from the input string are `%d`, `%e`, `%f`, `%g`, `%i`, `%o`, `%u`, and `%x`. When `sscanf` reads only numbers from the input string, the elements of the output matrix A are numbers.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a scalar `n`, `sscanf` reads at most `n` numbers from the input string. The output matrix is a column vector with one element for each number read from the input.

When the `size` argument is a matrix `[m,n]`, `sscanf` reads at most `(m*n)` numbers from the input string. The output matrix contains at most `m` rows and `n` columns. `sscanf` fills the output matrix in column order, using as many columns as it needs to contain all the numbers read from the input. Any unfilled elements in the final column contain zeros.

Output Characteristics: Only Character Values Read

The format characters that cause `sscanf` to read characters and strings from the input string are `%c` and `%s`. When `sscanf` reads only characters and strings from the input string, the elements of the output matrix A are characters. When `sscanf` reads a string from the input, the output matrix includes one element for each character in the string.

When there is no `size` argument or the `size` argument is `inf`, `sscanf` reads to the end of the input string. The output matrix is a row vector with one element for each character read from the input.

When the `size` argument is a scalar `n`, `sscanf` reads at most `n` character or string values from the input string. The output matrix is a row vector with one element for each character read from the input. When string values are read from the input, the output matrix can contain more than `n` columns.

When the `size` argument is a matrix `[m,n]`, `sscanf` reads at most `(m*n)` character or string values from the input string. The output
matrix contains at most \( m \) rows. \texttt{sscanf} fills the output matrix in column order, using as many columns as it needs to contain all the characters read from the input. When string values are read from the input, the output matrix can contain more than \( n \) columns. Any unfilled elements in the final column contain \texttt{char(0)}.

**Output Characteristics: Both Numeric and Character Values Read**

When \texttt{sscanf} reads a combination of numbers and either characters or strings from the input string, the elements of the output matrix \( A \) are numbers. This is true even when a format specifier such as \’\%*d \%s\’ tells MATLAB to ignore numbers in the input string and output only characters or strings. When \texttt{sscanf} reads a string from the input, the output matrix includes one element for each character in the string. All characters are converted to their numeric equivalents in the output matrix.

When there is no \texttt{size} argument or the \texttt{size} argument is \texttt{inf}, \texttt{sscanf} reads to the end of the input string. The output matrix is a column vector with one element for each character read from the input.

When the \texttt{size} argument is a scalar \( n \), \texttt{sscanf} reads at most \( n \) number, character, or string values from the input string. The output matrix contains at most \( n \) rows. \texttt{sscanf} fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than one column. Any unfilled elements in the final column contain zeros.

When the \texttt{size} argument is a matrix \([m,n]\), \texttt{sscanf} reads at most \((m*n)\) number, character, or string values from the input string. The output matrix contains at most \( m \) rows. \texttt{sscanf} fills the output matrix in column order, using as many columns as it needs to represent all the numbers and characters read from the input. When string values are read from the input, the output matrix can contain more than \( n \) columns. Any unfilled elements in the final column contain zeros.
Note This section applies only when sscanf actually reads a combination of numbers and either characters or strings from the input string. Even if the format string has both format characters that would result in numbers (such as %d) and format characters that would result in characters or strings (such as %s), sscanf might actually read only numbers or only characters or strings. If sscanf reads only numbers, see “Output Characteristics: Only Numeric Values Read” on page 2-3379. If sscanf reads only characters or strings, see “Output Characteristics: Only Character Values Read” on page 2-3379.

Examples

Example 1

The statements

```matlab
s = '2.7183 3.1416';
A = sscanf(s, '%f')
```

create a two-element vector containing poor approximations to e and pi.

Example 2

When using the %i conversion specifier, sscanf reads data starting with 0 as base 8 and returns the converted value as signed:

```matlab
sscanf('-010', '%i')
an =
-8
```

When using %o, on the other hand, sscanf returns the converted value as unsigned:

```matlab
sscanf('-010', '%o')
an =
4.2950e+009
```

Example 3

Create character array A representing both character and numeric data:
A = ['abc 46 6 ghi'; 'def 78 9 jkl']
A =
    abc 46 6 ghi
    def 78 9 jkl

Read A into 2-by-N matrix B, ignoring the character data. As stated in the Description section, `sscanf` reads the characters in A in column order, filling matrix B in column order:

\[
B = \text{sscanf}(A, '%*s \%d \%d %*s', [2, inf])
\]
B =
476
869

If you want `sscanf` to return the numeric data in B in the same order as in A, you can use this technique:

\[
\text{for } k = 1:2
\quad C(k,:) = \text{sscanf}(A(k,:), '%*s \%d \%d %*s', [1, inf]);
\end{eqnarray}
\]

C
C =
46 6
7 89

**See Also**
eval, sprintf, textread
**Purpose**

Stairstep graph

**GUI Alternatives**

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see *Plotting Tools — Interactive Plotting* in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

```
stairs(Y)
stairs(X,Y)
stairs(...,LineSpec)
stairs(...,'PropertyName',propertyvalue)
stairs(axes_handle,...)
h = stairs(...)
[xb,yb] = stairs(Y,...)
hlines = stairs('v6',...)
```

**Description**

Stairstep graphs are useful for drawing time-history graphs of digitally sampled data.

```
stairs(Y) draws a stairstep graph of the elements of Y, drawing one line per column for matrices. The axes ColorOrder property determines the color of the lines.
```

When Y is a vector, the x-axis scale ranges from 1 to `length(Y)`. When Y is a matrix, the x-axis scale ranges from 1 to the number of rows in Y.

```
stairs(X,Y) plots the elements in Y at the locations specified in X.
```

X must be the same size as Y or, if Y is a matrix, X can be a row or a column vector such that

```
length(X) = size(Y,1)
```
stairs

stairs(...,LineSpec) specifies a line style, marker symbol, and color for the graph. (See LineSpec for more information.)

stairs(...,'PropertyName',propertyvalue) creates the stairstep graph, applying the specified property settings. See Stairseries properties for a description of properties.

stairs(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes object (gca).

h = stairs(...) returns the handles of the stairseries objects created (one per matrix column).

[xb,yb] = stairs(Y,...) does not draw graphs, but returns vectors xb and yb such that plot(xb,yb) plots the stairstep graph.

Backward-Compatible Version

hlines = stairs('v6',...) returns the handles of line objects instead of stairseries objects for compatibility with MATLAB 6.5 and earlier.

Note The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Examples

Create a stairstep plot of a sine wave.

\[
\begin{align*}
x &= \text{linspace}(-2\cdot\pi,2\cdot\pi,40) \\
stairs(x,\sin(x))
\end{align*}
\]
See Also

bar, hist, stem

“Discrete Data Plots” on page 1-96 for related functions

Stairseries Properties for property descriptions
Stairseries Properties

**Purpose**
Define stairseries properties

**Modifying Properties**
You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default property values for stairseries objects.

See Plot Objects for information on stairseries objects.

**Stairseries Property Descriptions**
This section provides a description of properties. Curly braces `{}` enclose default values.

**Annotation**

hg.Annotation object Read Only

*Control the display of stairseries objects in legends.* The Annotation property enables you to specify whether this stairseries object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the stairseries object is displayed in a figure legend:

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Include the stairseries object in a legend as one entry, but not its children objects</td>
</tr>
</tbody>
</table>
### Stairseries Properties

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>Do not include the stairseries or its children in a legend (default)</td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the stairseries as separate entries in the legend</td>
</tr>
</tbody>
</table>

**Setting the IconDisplayStyle Property**

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

**Using the IconDisplayStyle Property**

See “Controlling Legends” for more information and examples.

**BeingDeleted**

on | {off} Read Only

*This object is being deleted.* The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to on when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object’s `BeingDeleted` property before acting.
BusyAction

cancel | {queue}

Callback routine interruption. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- cancel — Discard the event that attempted to execute a second callback routine.
- queue — Queue the event that attempted to execute a second callback routine until the current callback finishes.

ButtonDownFcn

string or function handle

Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.

See the figure’s SelectionType property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
• A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

The expression executes in the MATLAB workspace.
See Function Handle Callbacks for information on how to use function handles to define the callbacks.

Children
array of graphics object handles

*Children of this object.* The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object's HandleVisibility property is set to callback or off, its handle does not show up in this object's Children property unless you set the root ShowHiddenHandles property to on:

```
set(0,'ShowHiddenHandles','on')
```

Clipping
{on} | off

*Clipping mode.* MATLAB clips graphs to the axes plot box by default. If you set Clipping to off, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set hold to on, freeze axis scaling (axis manual), and then create a larger plot object.

Color
ColorSpec
**Color of the object.** A three-element RGB vector or one of the MATLAB predefined names, specifying the object’s color.

See the `ColorSpec` reference page for more information on specifying color.

**CreateFcn**

string or function handle

*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```matlab
area(y, 'CreateFcn', @CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

**DeleteFcn**

string or function handle

*Callback executed during object deletion.* A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying
the object’s properties so the callback routine can query these values.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which can be queried using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the BeingDeleted property for related information.

**DisplayName**

string (default is empty string)

*String used by legend for this stairseries object.* The legend function uses the string defined by the DisplayName property to label this stairseries object in the legend.

- If you specify string arguments with the legend function, DisplayName is set to this stairseries object’s corresponding string and that string is used for the legend.

- If DisplayName is empty, legend creates a string of the form, ['data' 'n'], where n is the number assigned to the object based on its location in the list of legend entries. However, legend does not set DisplayName to this string.

- If you edit the string directly in an existing legend, DisplayName is set to the edited string.

- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.

- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See “Controlling Legends” for more examples.
**EraseMode**

{normal} | none | xor | background

**Erase mode.** This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with **EraseMode none**, you cannot print these objects because MATLAB stores no information about their former locations.

- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes **Color** property is set to none). That is, it isn’t erased correctly if there are objects behind it.

- **background** — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes **Color** property is set to none). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

**Printing with Nonnormal Erase Modes**
MATLAB always prints figures as if the EraseMode of all objects is normal. This means graphics objects created with EraseMode set to none, xor, or background can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes Color property. Set the figure background color with the figure Color property.

You can use the MATLAB getframe command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

**HandleVisibility**

{on} | callback | off

*Control access to object’s handle by command-line users and GUIs.* This property determines when an object’s handle is visible in its parent’s list of children. HandleVisibility is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- **on** — Handles are always visible when HandleVisibility is on.
- **callback** — Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- **off** — Setting HandleVisibility to off makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.
Functions Affected by Handle Visibility

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle’s visibility is restricted using `callback` or `off`, the object’s handle does not appear in its parent’s `Children` property, figures do not appear in the root’s `CurrentFigure` property, objects do not appear in the root’s `CallbackObject` property or in the figure’s `CurrentObject` property, and axes do not appear in their parent’s `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object’s handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
**Stairseries Properties**

**HitTest**

{on} | off

*Selectable by mouse click.* HitTest determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If HitTest is off, clicking this object selects the object below it (which is usually the axes containing it).

**HitTestArea**

on | {off}

*Select the object by clicking lines or area of extent.* This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When HitTestArea is off, you must click the object’s lines or markers (excluding the baseline, if any) to select the object. When HitTestArea is on, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).

**Interruptible**

{on} | off

*Callback routine interruption mode.* The Interruptible property controls whether an object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the `ButtonDownFcn` property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

2-3395
Setting `Interruptible` to on allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**LineStyle**
```
{-} | - | : | - . | none
```

*Line style.* This property specifies the line style of the object. Available line styles are shown in the following table.

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>--</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

**LineWidth**
```
scalar
```

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = \( \frac{1}{72} \) inch). The default `LineWidth` is 0.5 points.

**Marker**
```
character (see table)
```

*Marker symbol.* The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the
Marker property independently from the LineStyle property. Supported markers include those shown in the following table.

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>o</td>
<td>Circle</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

**MarkerEdgeColor**

ColorSpec | none | {auto}

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the Color property.

**MarkerFaceColor**

ColorSpec | {none} | auto
**Stairseries Properties**

*Marker face color.* The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or to the figure color if the axes `Color` property is set to `none` (which is the factory default for axes objects).

**MarkerSize**

size in points

*Marker size.* A scalar specifying the size of the marker in points. The default value for `MarkerSize` is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

**Parent**

handle of parent axes, hggroup, or hgtransform

*Parent of this object.* This property contains the handle of the object’s parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

**Selected**

on | {off}

*Is object selected?* When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the `SelectionHighlight` property is also on (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

**SelectionHighlight**

{on} | off
Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

Tag

string

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define Tag as any string.

For example, you might create an areaseries object and set the Tag property.

\[
t = \text{area}(Y, 'Tag', 'area1')
\]

When you want to access objects of a given type, you can use findobj to find the object's handle. The following statement changes the FaceColor property of the object whose Tag is area1.

\[
\text{set} (\text{findobj}('Tag', 'area1'), 'FaceColor', 'red')
\]

Type

string (read only)

Type of graphics object. This property contains a string that identifies the class of the graphics object. For stairseries objects, Type is 'hggroup'. The following statement finds all the hggroup objects in the current axes object.

\[
t = \text{findobj}(\text{gca}, 'Type', 'hggroup');
\]
Stairseries Properties

**UIContextMenu**

handle of a uicontextmenu object

*Associate a context menu with this object.* Assign this property the handle of a uicontextmenu object created in the object’s parent figure. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

**UserData**

array

*User-specified data.* This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

**Visible**

{on} | off

*Visibility of this object and its children.* By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s `Visible` property is set to off. Setting an object’s `Visible` property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.

**XData**

array

*X-axis location of stairs.* The stairs function uses `XData` to label the x-axis. `XData` can be either a matrix equal in size to `YData` or a vector equal in length to the number of rows in `YData`. That is, `length(XData) == size(YData,1)`.

If you do not specify `XData` (i.e., the input argument `x`), the `stairs` function uses the indices of `YData` to create the stairstep graph. See the `XDataMode` property for related information.
**Stairseries Properties**

**XDataMode**

{auto} | manual

*Use automatic or user-specified x-axis values.* If you specify XData (by setting the XData property or specifying the x input argument), MATLAB sets this property to manual and uses the specified values to label the x-axis.

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks to 1:size(YData,1) or to the column indices of the ZData, overwriting any previous values for XData.

**XDataSource**

string (MATLAB variable)

*Link XData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change XData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
**YData**

scalar, vector, or matrix

*Stairs plot data.* YData contains the data plotted in the stairstep graph. Each value in YData is represented by a marker in the stairstep graph. If YData is a matrix, the stairs function creates a line for each column in the matrix.

The input argument y in the stairs function calling syntax assigns values to YData.

**YDataSource**

string (MATLAB variable)

*Link YData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change YData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

---

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Purpose: Start timer(s) running

Syntax: start(obj)

Description: start(obj) starts the timer running, represented by the timer object, obj. If obj is an array of timer objects, start starts all the timers. Use the timer function to create a timer object.

start sets the Running property of the timer object, obj, to 'on', initiates TimerFcn callbacks, and executes the StartFcn callback.

The timer stops running if one of the following conditions apply:

- The first TimerFcn callback completes, if ExecutionMode is 'singleShot'.
- The number of TimerFcn callbacks specified in TasksToExecute have been executed.
- The stop(obj) command is issued.
- An error occurred while executing a TimerFcn callback.

See Also: timer, stop
**Purpose**

Start timer(s) running at specified time

**Syntax**

```
startat(obj,time)
startat(obj,S)
startat(obj,S,pivotyear)
startat(obj,Y,M,D)
startat(obj,[Y,M,D])
startat(obj,Y,M,D,H,MI,S)
startat(obj,[Y,M,D,H,MI,S])
```

**Description**

`startat(obj,time)` starts the timer running, represented by the timer object `obj`, at the time specified by the serial date number `time`. If `obj` is an array of timer objects, `startat` starts all the timers running at the specified time. Use the `timer` function to create the timer object.

`startat` sets the `Running` property of the timer object, `obj`, to 'on', initiates `TimerFcn` callbacks, and executes the `StartFcn` callback.

The serial date number, `time`, indicates the number of days that have elapsed since 1-Jan-0000 (starting at 1). See `datenum` for additional information about serial date numbers.

`startat(obj,S)` starts the timer running at the time specified by the date string `S`. The date string must use date format 0, 1, 2, 6, 13, 14, 15, 16, or 23, as defined by the `datestr` function. Date strings with two-character years are interpreted to be within the 100 years centered on the current year.

`startat(obj,S,pivotyear)` uses the specified pivot year as the starting year of the 100-year range in which a two-character year resides. The default pivot year is the current year minus 50 years.

`startat(obj,Y,M,D)` `startat(obj,[Y,M,D])` start the timer at the year (`Y`), month (`M`), and day (`D`) specified. `Y`, `M`, and `D` must be arrays of the same size (or they can be a scalar).

`startat(obj,Y,M,D,H,MI,S)` `startat(obj,[Y,M,D,H,MI,S])` start the timer at the year (`Y`), month (`M`), day (`D`), hour (`H`), minute (`MI`), and second (`S`) specified. `Y`, `M`, `D`, `H`, `MI`, and `S` must be arrays of the same size (or they can be a scalar). Values outside the normal range of each array
are automatically carried to the next unit (for example, month values greater than 12 are carried to years). Month values less than 1 are set to be 1; all other units can wrap and have valid negative values.

The timer stops running if one of the following conditions apply:

- The number of TimerFcn callbacks specified in TasksToExecute have been executed.
- The stop(obj) command is issued.
- An error occurred while executing a TimerFcn callback.

**Examples**

This example uses a timer object to execute a function at a specified time.

```matlab
t1=timer('TimerFcn','disp('''it is 10 o''''clock'''));
startat(t1,'10:00:00');
```

This example uses a timer to display a message when an hour has elapsed.

```matlab
t2=timer('TimerFcn','disp('''It has been an hour now.''''));
startat(t2,now+1/24);
```

**See Also**
datenum, datestr, now, timer, start, stop
**Purpose**
Startup M-file for user-defined options

**Syntax**
startup

**Description**
startup automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`, when the MATLAB program starts. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on the search path MATLAB uses. You can create a `startup.m` file in your own startup directory for MATLAB. The file can include physical constants, defaults for Handle Graphics properties, engineering conversion factors, or anything else you want predefined in your workspace.

**Algorithm**
At startup, MATLAB only runs `matlabrc.m`. However, `matlabrc.m` contains the statements

```matlab
if exist('startup')==2
    startup
end
```

that run `startup.m`. You can extend this process to create additional startup M-files, if needed.

**See Also**
finish, `matlabrc`, `matlabroot`, path, quit, `userpath`

See “Specifying Startup Options Using the Startup File for the MATLAB Program, `startup.m`” and Preferences in the MATLAB Desktop Tools and Development Environment documentation.
Purpose

Standard deviation

Syntax

s = std(X)
s = std(X,flag)
s = std(X,flag,dim)

Definition

There are two common textbook definitions for the standard deviation $s$ of a data vector $X$.

\begin{equation}
(1) \quad s = \left( \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 \right)^{1/2}
\end{equation}

\begin{equation}
(2) \quad s = \left( \frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{x})^2 \right)^{1/2}
\end{equation}

where

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

and $n$ is the number of elements in the sample. The two forms of the equation differ only in $n - 1$ versus $n$ in the divisor.

Description

$s = \text{std}(X)$, where $X$ is a vector, returns the standard deviation using (1) above. The result $s$ is the square root of an unbiased estimator of the variance of the population from which $X$ is drawn, as long as $X$ consists of independent, identically distributed samples.

If $X$ is a matrix, $\text{std}(X)$ returns a row vector containing the standard deviation of the elements of each column of $X$. If $X$ is a multidimensional array, $\text{std}(X)$ is the standard deviation of the elements along the first nonsingleton dimension of $X$. 
s = std(X,flag) for flag = 0, is the same as std(X). For flag = 1, std(X,1) returns the standard deviation using (2) above, producing the second moment of the set of values about their mean.

s = std(X,flag,dim) computes the standard deviations along the dimension of X specified by scalar dim. Set flag to 0 to normalize Y by n-1; set flag to 1 to normalize by n.

**Examples**

For matrix X

\[
X =
\begin{bmatrix}
1 & 5 & 9 \\
7 & 15 & 22
\end{bmatrix}
\]

s = std(X,0,1)

\[
s =
\begin{bmatrix}
4.2426 & 7.0711 & 9.1924
\end{bmatrix}
\]

s = std(X,0,2)

\[
s =
\begin{bmatrix}
4.000 \\
7.5056
\end{bmatrix}
\]

**See Also**
corrcoef, cov, mean, median, var
Purpose

Standard deviation of timeseries data

Syntax

\[
\text{ts}_\text{std} = \text{std}(\text{ts})
\]
\[
\text{ts}_\text{std} = \text{std}(\text{ts}, \text{'PropertyName1'}, \text{PropertyValue1}, ...)
\]

Description

\( \text{ts}_\text{std} = \text{std}(\text{ts}) \) returns the standard deviation of the time-series data. When \( \text{ts.Data} \) is a vector, \( \text{ts}_\text{std} \) is the standard deviation of \( \text{ts.Data} \) values. When \( \text{ts.Data} \) is a matrix, \( \text{ts}_\text{std} \) is the standard deviation of each column of \( \text{ts.Data} \) (when \( \text{IsTimeFirst} \) is true and the first dimension of \( \text{ts} \) is aligned with time). For the N-dimensional \( \text{ts.Data} \) array, \text{std} always operates along the first nonsingleton dimension of \( \text{ts.Data} \).

\( \text{ts}_\text{std} = \text{std}(\text{ts}, \text{'PropertyName1'}, \text{PropertyValue1}, ...) \)
specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.

- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

- 'Weighting' property has two possible values, 'none' (default) or 'time'. When you specify 'time', larger time values correspond to larger weights.

Examples

1 Load a 24-by-3 data array.

   \[
   \text{load count.dat}
   \]

2 Create a timeseries object with 24 time values.

   \[
   \text{count_ts = timeseries(count,1:24,'Name','CountPerSecond')}
   \]
Calculate the standard deviation of each data column for this `timeseries` object.

```matlab
std(count_ts)
```

```
ans =

    25.3703    41.4057    68.0281
```

The standard deviation is calculated independently for each data column in the `timeseries` object.

**See Also**

- `iqr (timeseries)`
- `mean (timeseries)`
- `median (timeseries)`
- `var (timeseries)`
- `timeseries`
Purpose
Plot discrete sequence data

GUI Alternatives
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax
stem(Y)
stem(X,Y)
stem(...,'fill')
stem(...,LineSpec)
stem(axes_handle,...)
h = stem(...)
hlines = stem('v6',...)

Description
A two-dimensional stem plot displays data as lines extending from a baseline along the x-axis. A circle (the default) or other marker whose y-position represents the data value terminates each stem.

stem(Y) plots the data sequence Y as stems that extend from equally spaced and automatically generated values along the x-axis. When Y is a matrix, stem plots all elements in a row against the same x value.

stem(X,Y) plots X versus the columns of Y. X and Y must be vectors or matrices of the same size. Additionally, X can be a row or a column vector and Y a matrix with length(X) rows.

stem(...,'fill') specifies whether to color the circle at the end of the stem.

stem(...,LineSpec) specifies the line style, marker symbol, and color for the stem and top marker (the baseline is not affected). See LineSpec for more information.
stem

stem(axes_handle,...) plots into the axes object with the handle
axes_handle instead of into the current axes object (gca).

h = stem(...) returns a vector of stemseries object handles in h, one
handle per column of data in Y.

Backward-Compatible Version

hlines = stem('v6',...) returns the handles of line objects instead
of stemseries objects for compatibility with MATLAB 6.5 and earlier.

hlines contains the handles to three line graphics objects:

- hlines(1) — The marker symbol at the top of each stem
- hlines(2) — The stem line
- hlines(3) — The baseline handle

Note The v6 option enables users of Version 7.x of MATLAB to create
FIG-files that previous versions can open. It is obsolete and will be
removed in a future version of MATLAB.

Examples

Single Series of Data

This example creates a stem plot representing the cosine of 10 values
linearly spaced between 0 and 2π. Note that the line style of the
baseline is set by first getting its handle from the stemseries object’s
BaseLine property.

```matlab
    t = linspace(-2*pi,2*pi,10);
    h = stem(t,cos(t),'fill','--');
    set(get(h,'BaseLine'),'LineStyle','--');
    set(h,'MarkerFaceColor','red')
```
The following diagram illustrates the parent-child relationship in the previous stem plot. Note that the stemseries object contains two line objects used to draw the stem lines and the end markers. The baseline is a separate line object.
If you do not want the baseline to show, you can remove it with the following command:

```matlab
delete(get(stem_handle,'Baseline'))
```

where `stem_handle` is the handle for the stemseries object. You can use similar code to change the color or style of the baseline, specifying any line property and value, for example,

```matlab
set(get(stem_handle,'Baseline'),'LineWidth',3)
```

**Two Series of Data on One Graph**

The following example creates a stem plot from a two-column matrix. In this case, the `stem` function creates two stemseries objects, one of each column of data. Both objects' handles are returned in the output argument `h`.

- `h(1)` is the handle to the stemseries object plotting the expression `exp(-.07*x).*cos(x)`.
- `h(2)` is the handle to the stemseries object plotting the expression `exp(.05*x).*cos(x)`.

```matlab
x = 0:25;
y = [exp(-.07*x).*cos(x);exp(.05*x).*cos(x)]';
h = stem(x,y);
```
The following diagram illustrates the parent-child relationship in the previous stem plot. Note that each column in the input matrix y results in the creation of a stemseries object, which contains two line objects (one for the stems and one for the markers). The baseline is shared by both stemseries objects.
See Also  
bar, plot, stairs

Stemseries properties for property descriptions
**Purpose**

Plot 3-D discrete sequence data

**GUI Alternatives**

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

```plaintext
stem3(Z)
stem3(X,Y,Z)
stem3(...,'fill')
stem3(...,LineSpec)
```

```plaintext
h = stem3(...)
hlines = stem3('v6',...)
```

**Description**

Three-dimensional stem plots display lines extending from the *x*- *y* plane. A circle (the default) or other marker symbol whose *z*-position represents the data value terminates each stem.

- `stem3(Z)` plots the data sequence *Z* as stems that extend from the *x*- *y* plane. *x* and *y* are generated automatically. When *Z* is a row vector, `stem3` plots all elements at equally spaced *x* values against the same *y* value. When *Z* is a column vector, `stem3` plots all elements at equally spaced *y* values against the same *x* value.

- `stem3(X,Y,Z)` plots the data sequence *Z* at values specified by *X* and *Y*. *X*, *Y*, and *Z* must all be vectors or matrices of the same size.

- `stem3(...,'fill')` specifies whether to color the interior of the circle at the end of the stem.

- `stem3(...,LineSpec)` specifies the line style, marker symbol, and color for the stems. See `LineSpec` for more information.

- `h = stem3(...)` returns handles to stemseries graphics objects.
**Backward-Compatible Version**

`hlines = stem3('v6',...)` returns the handles of line objects instead of stemseries objects for compatibility with MATLAB 6.5 and earlier.

**Note** The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

**Examples**

Create a three-dimensional stem plot to visualize a function of two variables.

```matlab
X = linspace(0,1,10);
Y = X./2;
Z = sin(X) + cos(Y);
stem3(X,Y,Z,'fill')
view(-25,30)
```
See Also

bar, plot, stairs, stem

“Discrete Data Plots” on page 1-96 for related functions

Stemseries Properties for descriptions of properties

Three-Dimensional Stem Plots for more examples
Stemseries Properties

**Purpose**
Define stemseries properties

**Modifying Properties**
You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

Note that you cannot define default properties for stemseries objects.

See Plot Objects for information on stemseries objects.

**Stemseries Property Descriptions**
This section provides a description of properties. Curly braces {} enclose default values.

**Annotation**

hg.Annotation object Read Only

*Control the display of stemseries objects in legends.* The Annotation property enables you to specify whether this stemseries object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the stemseries object is displayed in a figure legend:

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Include the stemseries object in a legend as one entry, but not its children objects</td>
</tr>
<tr>
<td>off</td>
<td>Do not include the stemseries or its children in a legend (default)</td>
</tr>
<tr>
<td>children</td>
<td>Include only the children of the stemseries as separate entries in the legend</td>
</tr>
</tbody>
</table>
Setting the IconDisplayStyle Property

These commands set the IconDisplayStyle of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

BaseLine
handle of baseline

*Handle of the baseline object.* This property contains the handle of the line object used as the baseline. You can set the properties of this line using its handle. For example, the following statements create a stem plot, obtain the handle of the baseline from the stemseries object, and then set line properties that make the baseline a dashed, red line.

```matlab
stem_handle = stem(randn(10,1));
baseline_handle = get(stem_handle,'BaseLine');
set(baseline_handle,'LineStyle','--','Color','red')
```

BaseValue
`y`-axis value

*Y-axis value where baseline is drawn.* You can specify the value along the `y`-axis at which the MATLAB software draws the baseline.

BeingDeleted
on | {off} Read Only
This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object’s `BeingDeleted` property before acting.

**BusyAction**

```matlab
cancel | {queue}
```

*Callback routine interruption.* The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.

**ButtonDownFcn**

```matlab
string or function handle
```
Button press callback function. A callback that executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the HitTestArea property for information about selecting objects of this type.

See the figure’s SelectionType property to determine if modifier keys were also pressed.

This property can be
- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See “Function Handle Callbacks” for information on how to use function handles to define the callbacks.

Children
array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

Note that if a child object’s HandleVisibility property is set to callback or off, its handle does not show up in this object’s Children property unless you set the root ShowHiddenHandles property to on:

    set(0,'ShowHiddenHandles','on')

Clipping
{on} | off
**Clipping mode.** MATLAB clips graphs to the axes plot box by default. If you set Clipping to off, portions of graphs can be displayed outside the axes plot box. This can occur if you create a plot object, set hold to on, freeze axis scaling (axis manual), and then create a larger plot object.

**Color**

*ColorSpec*

*Color of stem lines.* A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the ColorSpec reference page for more information on specifying color.

For example, the following statement would produce a stem plot with red lines.

```matlab
h = stem(randn(10,1),'Color','r');
```

**CreateFcn**

*string or function handle*

*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```matlab
area(y,'CreateFcn',@CallbackFcn)
```

where @CallbackFcn is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.
The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

**DeleteFcn**

string or function handle

*Callback executed during object deletion.* A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

**DisplayName**

string (default is empty string)

*String used by legend for this stemseries object.* The `legend` function uses the string defined by the `DisplayName` property to label this stemseries object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this stemseries object’s corresponding string and that string is used for the legend.

- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where `n` is the number assigned to the object
Stemseries Properties

Based on its location in the list of legend entries. However, legend does not set DisplayName to this string.

- If you edit the string directly in an existing legend, DisplayName is set to the edited string.
- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.
- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See “Controlling Legends” for more examples.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- normal — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- none — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with EraseMode none, you cannot print these objects because MATLAB stores no information about their former locations.
- xor — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of
the screen behind it and it is correctly colored only when it is
over the axes background color (or the figure background color
if the axes Color property is set to none). That is, it isn’t erased
correctly if there are objects behind it.

• background — Erase the graphics objects by redrawing them
in the axes background color, (or the figure background color
if the axes Color property is set to none). This damages other
graphics objects that are behind the erased object, but the
erased object is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the EraseMode of all objects
is normal. This means graphics objects created with EraseMode
set to none, xor, or background can look different on screen than
on paper. On screen, MATLAB can mathematically combine
layers of colors (e.g., performing an XOR on a pixel color with that
of the pixel behind it) and ignore three-dimensional sorting to
obtain greater rendering speed. However, these techniques are
not applied to the printed output.

Set the axes background color with the axes Color property. Set
the figure background color with the figure Color property.

You can use the MATLAB getframe command or other screen
capture applications to create an image of a figure containing
nonnormal mode objects.

HandleVisibility
{on} | callback | off

Control access to object’s handle by command-line users and GUIs.
This property determines when an object’s handle is visible in
its parent’s list of children. HandleVisibility is useful for
preventing command-line users from accidentally accessing
objects that you need to protect for some reason.
Stemseries Properties

- **on** — Handles are always visible when HandleVisibility is on.
- **callback** — Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- **off** — Setting HandleVisibility to off makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle’s visibility is restricted using **callback** or **off**, the object’s handle does not appear in its parent’s **Children** property, figures do not appear in the root’s **CurrentFigure** property, objects do not appear in the root’s **CallbackObject** property or in the figure’s **CurrentObject** property, and axes do not appear in their parent’s **CurrentAxes** property.

Overriding Handle Visibility

You can set the root **ShowHiddenHandles** property to on to make all handles visible regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties). See also `findall`. 

2-3428
Handle Validity

Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties and pass it to any function that operates on handles.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

HitTest

{on} | off

*Selectable by mouse click.* HitTest determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If HitTest is off, clicking this object selects the object below it (which is usually the axes containing it).

HitTestArea

on | {off}

*Select the object by clicking lines or area of extent.* This property enables you to select plot objects in two ways:

- Select by clicking lines or markers (default).
- Select by clicking anywhere in the extent of the plot.

When HitTestArea is off, you must click the object’s lines or markers (excluding the baseline, if any) to select the object. When HitTestArea is on, you can select this object by clicking anywhere within the extent of the plot (i.e., anywhere within a rectangle that encloses it).
**Stemseries Properties**

**Interruptible**

\{on\} | off

*Callback routine interruption mode.* The Interruptible property controls whether an object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the BusyAction property for related information.

Setting Interruptible to on allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**LineStyle**

\{-\} | – | : | -. | none

*Line style.* This property specifies the line style of the object. Available line styles are shown in the following table.

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>--</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>
You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

### LineWidth

*scalar*

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = \(\frac{1}{72}\) inch). The default `LineWidth` is 0.5 points.

### Marker

*character (see table)*

*Marker symbol.* The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

<table>
<thead>
<tr>
<th><strong>Marker Specifier</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>o</td>
<td>Circle</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
</tbody>
</table>
**Stemseries Properties**

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

**MarkerEdgeColor**

ColorSpec | none | {auto}

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the Color property.

**MarkerFaceColor**

ColorSpec | {none} | auto

*Marker face color.* The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or to the figure color if the axes Color property is set to none (which is the factory default for axes objects).

**MarkerSize**

size in points

*Marker size.* A scalar specifying the size of the marker in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the ‘.’ symbol) at one-third the specified size.

**Parent**

handle of parent axes, hggroup, or hgtransform
*Parent of this object.* This property contains the handle of the object’s parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

**Selected**

```
on | {off}
```

*Is object selected?* When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

**SelectionHighlight**

```
{on} | off
```

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

**Tag**

```
string
```

*User-specified object label.* The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks.

For example, you might create a stemseries object and set the Tag property:
Stemseries Properties

```matlab
t = stem(Y,'Tag','stem1')
```

When you want to access the stemseries object, you can use `findobj` to find the stemseries object’s handle. The following statement changes the `MarkerFaceColor` property of the object whose `Tag` is `stem1`.

```matlab
set(findobj('Tag','stem1'),'MarkerFaceColor','red')
```

**Type**

string (read only)

*Type of graphics object.* This property contains a string that identifies the class of the graphics object. For stemseries objects, `Type` is `'hggroup'`. The following statement finds all the `hggroup` objects in the current axes object.

```matlab
t = findobj(gca,'Type','hggroup');
```

**UIContextMenu**

handle of a `uicontextmenu` object

*Associate a context menu with this object.* Assign this property the handle of a `uicontextmenu` object created in the object’s parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

**UserData**

array

*User-specified data.* This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

**Visible**

{on} | off
Visibility of this object and its children. By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s Visible property is set to off. Setting an object’s Visible property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.

XData array

X-axis location of stems. The stem function draws an individual stem at each x-axis location in the XData array. XData can be either a matrix equal in size to YData or a vector equal in length to the number of rows in YData. That is, length(XData) == size(YData,1). XData does not need to be monotonically increasing.

If you do not specify XData (i.e., the input argument x), the stem function uses the indices of YData to create the stem plot. See the XDataMode property for related information.

XDataMode
{auto} | manual

Use automatic or user-specified x-axis values. If you specify XData (by setting the XData property or specifying the x input argument), MATLAB sets this property to manual and uses the specified values to label the x-axis.

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks to 1:size(YData,1) or to the column indices of the ZData, overwriting any previous values for XData.

XDataSource string (MATLAB variable)
**Stemseries Properties**

*Link XData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change XData.

You can use the refreshdata function to force an update of the object's data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

---

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

**YData**

scalar, vector, or matrix

*Stem plot data.* YData contains the data plotted as stems. Each value in YData is represented by a marker in the stem plot. If YData is a matrix, MATLAB creates a series of stems for each column in the matrix.

The input argument y in the stem function calling syntax assigns values to YData.

**YDataSource**

string (MATLAB variable)
**Stemseries Properties**

*Link YData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change YData.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

---

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

---

**ZData**

vector of coordinates

*Z-coordinates.* A data defining the stems for 3-D stem graphs. XData and YData (if specified) must be the same size.

**ZDataSource**

string (MATLAB variable)

*Link ZData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the ZData.
MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change ZData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Purpose
Stop timer(s)

Syntax
stop(obj)

Description
stop(obj) stops the timer, represented by the timer object, obj. If obj is an array of timer objects, the stop function stops them all. Use the timer function to create a timer object.

The stop function sets the Running property of the timer object, obj, to 'off', halts further TimerFcn callbacks, and executes the StopFcn callback.

See Also
timer, start
Purpose
Stop asynchronous read and write operations

Syntax
stopasync(obj)

Description
stopasync(obj) stops any asynchronous read or write operation that is in progress for the serial port object, obj.

Remarks
You can write data asynchronously using the fprintf or fwrite function. You can read data asynchronously using the readasync function, or by configuring the ReadAsyncMode property to continuous. In-progress asynchronous operations are indicated by the TransferStatus property.

If obj is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:

- Its TransferStatus property is configured to idle.
- Its ReadAsyncMode property is configured to manual.
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the readasync function, or configure the ReadAsyncMode property to continuous, then the new data is appended to the existing data in the input buffer.

See Also

Functions
fprintf, fwrite, readasync

Properties
ReadAsyncMode, TransferStatus
Purpose
Convert string to double-precision value

Syntax
X = str2double('str')
X = str2double(C)

Description
X = str2double('str') converts the string str, which should be an ASCII character representation of a real or complex scalar value, to the MATLAB double-precision representation. The string can contain digits, a comma (thousands separator), a decimal point, a leading + or - sign, an e preceding a power of 10 scale factor, and an i for a complex unit.

If str does not represent a valid scalar value, str2double returns NaN.

X = str2double(C) converts the strings in the cell array of strings C to double precision. The matrix X returned will be the same size as C.

Examples
Here are some valid str2double conversions.

    str2double('123.45e7')
    str2double('123 + 45i')
    str2double('3.14159')
    str2double('2.7i - 3.14')
    str2double({'2.71' '3.1415'})
    str2double('1,200.34')

See Also
char, hex2num, num2str, str2num
Purpose
Construct function handle from function name string

Syntax
str2func('str')

Description
str2func('str') constructs a function handle `fh` for the function named in the string 'str'. The contents of `str` can be the name of a function M-file, or the name of an anonymous function.

You can create a function handle `fh` using any of the following four methods:

- Create a handle to a named function:

  ```matlab
  fh = @functionName;
  fh = str2func(functionName);
  ```

- Create a handle to an anonymous function:

  ```matlab
  fh = @(x)functionDef(x);
  fh = str2func('@(x)functionDef(x)');
  ```

You can create an array of function handles from strings by creating the handles individually with `str2func`, and then storing these handles in a cell array.

Remarks
Nested functions are not accessible to `str2func`. To construct a function handle for a nested function, you must use the function handle constructor, `@`.

Examples
Example 1
To convert the string, 'sin', into a handle for that function, type

```matlab
fh = str2func('sin')
fh =
    @sin
```
**Example 2**

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using `str2func`. The example below passes the variable, `funcname`, to function `makeHandle`, which then creates a function handle. Here is the function M-file:

```matlab
function fh = makeHandle(funcname)
    fh = str2func(funcname);
end
```

This is the code that calls `makeHandle` to construct the function handle:

```matlab
makeHandle('sin')
```

**Example 3**

To call `str2func` on a cell array of strings, use the `cellfun` function. This returns a cell array of function handles:

```matlab
fh_array = cellfun(@str2func, {'sin' 'cos' 'tan'}, ...
                 'UniformOutput', false);
```

```matlab
fh_array{2}(5)
```

**Example 4**

In the following example, the `myminbnd` function expects to receive either a function handle or string in the first argument. If you pass a string, `myminbnd` constructs a function handle from it using `str2func`, and then uses that handle in a call to `fminbnd`:

```matlab
function myminbnd(fhandle, lower, upper)
    if ischar(fhandle)
        disp 'converting function string to function handle ...'
        fhandle = str2func(fhandle);
    end
end
```
fminbnd(fhandle, lower, upper)

Whether you call myminbnd with a function handle or function name string, the function can handle the argument appropriately:

myminbnd('humps', 0.3, 1)
converting function string to function handle ...
ans =
   0.6370

Example 5

The dirByType function shown here creates an anonymous function called dirCheck. What the anonymous function does depends upon the value of the dirType argument passed in to the primary function. The example demonstrates one possible use of str2func with anonymous functions:

function dirByType(dirType)
    switch(dirType)
        case 'class', leadchar = '@';
        case 'package', leadchar = '+';
        otherwise disp('ERROR: Unrecognized type'), return;
    end

dirfile = @(fs)isdir(fs.name);
dirCheckStr = ['@(fs)strcmp(fs.name(1,1),''', leadchar, ''')'];
dirCheckFun = str2func(dirCheckStr);
s = dir; filecount = length(s);

for k=1:filecount
    fstruct = s(k);
    if dirfile(fstruct) && dirCheckFun(fstruct)
        fprintf('%s directory: %s
', dirType, fstruct.name)
    end
end

Generate a list of class and package directories:
dirByType('class')
class directory: @Point
class directory: @asset
class directory: @bond

dirByType('package')
package directory: +containers
package directory: +event
package directory: +mypkg

See Also
function_handle, func2str, functions
**str2mat**

**Purpose**
Form blank-padded character matrix from strings

**Syntax**

```matlab
S = str2mat(T1, T2, T3, ...)
```

**Description**

`S = str2mat(T1, T2, T3, ...)` forms the matrix `S` containing the text strings `T1, T2, T3, ...` as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, `Ti`, can itself be a string matrix. This allows the creation of arbitrarily large string matrices. Empty strings are significant.

**Note**
This routine will become obsolete in a future version. Use `char` instead.

**Remarks**

`str2mat` differs from `strvcat` in that empty strings produce blank rows in the output. In `strvcat`, empty strings are ignored.

**Examples**

```matlab
x = str2mat('36842', '39751', '38453', '90307');

whos x
```

```
Name       Size      Bytes  Class
-------    -------   ------  -----
   x        4x5     40  char array

x(2,3)
```

ans =

```
7
```

**See Also**

`char`, `strvcat`
**Purpose**
Convert string to number

**Syntax**

```matlab
x = str2num('str')
[x, status] = str2num('str')
```

**Description**

**Note** `str2num` uses the `eval` function to convert the input argument. Side effects can occur if the string contains calls to functions. Using `str2double` can avoid some of these side effects.

The following table shows several examples of valid inputs to `str2num`:

<table>
<thead>
<tr>
<th>String Input</th>
<th>Numeric Output</th>
<th>Output Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>'500'</td>
<td>500</td>
<td>1-by-1 scalar double</td>
</tr>
<tr>
<td>'500 250 125 67'</td>
<td>500, 250, 125, 67</td>
<td>1-by-4 row vector of double</td>
</tr>
<tr>
<td>'500; 250; 125; 62.5'</td>
<td>500.0000 250.0000 125.0000 62.5000</td>
<td>4-by-1 column vector of double</td>
</tr>
</tbody>
</table>
str2num

<table>
<thead>
<tr>
<th>String Input</th>
<th>Numeric Output</th>
<th>Output Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>'1 23 6 21; 53:56'</td>
<td>1 23 6 21</td>
<td>2-by-5 matrix of double</td>
</tr>
<tr>
<td></td>
<td>53 54 55 56</td>
<td></td>
</tr>
<tr>
<td>'12e-3 5.9e-3'</td>
<td>0.0120 0.0059</td>
<td>vector of double</td>
</tr>
<tr>
<td>'uint16(500)'</td>
<td>500</td>
<td>16–bit unsigned integer</td>
</tr>
</tbody>
</table>

If the input string does not represent a valid number or matrix, \texttt{str2num(str)} returns the empty matrix in \( x \).

\([x, \text{status}] = \texttt{str2num('str')}\) returns the status of the conversion in logical \( \text{status} \), where \( \text{status} \) equals logical 1 (true) if the conversion succeeds, and logical 0 (false) otherwise.

Space characters can be significant. For instance, \texttt{str2num('1+2i')} and \texttt{str2num('1 + 2i')} produce \( x = 1+2i \), while \texttt{str2num('1 +2i')} produces \( x = [1 2i] \). You can avoid these problems by using the \texttt{str2double} function.

**Examples**

Input a character string that contains a single number. The output is a scalar double:

\[
\begin{align*}
A &= \texttt{str2num('500')} \\
A &= 500 \\
\text{class}(A) &= \text{ans} = \text{double}
\end{align*}
\]

Repeat this operation, but this time using an unsigned 16–bit integer:

\[
\begin{align*}
A &= \texttt{str2num('uint16(500)')} \\
A &= 500 \\
\text{class}(A) &= \text{double}
\end{align*}
\]
ans =
    uint16

Try three different ways of specifying a row vector. Each returns the same answer:

```matlab
str2num('2 4 6 8') % Separate with spaces.
ans =
    2    4    6    8

str2num('2,4,6,8') % Separate with commas.
ans =
    2    4    6    8

str2num('[2 4 6 8]') % Enclose in brackets.
ans =
    2    4    6    8
```

Note that the first two of these commands do not need the MATLAB square bracket operator to create a matrix. The `str2num` function inserts the brackets for you if they are needed.

Use a column vector this time:

```matlab
str2num('2; 4; 6; 8')
ans =
    2
    4
    6
    8
```

And now a 2-by-2 matrix:

```matlab
str2num('2 4; 6 8')
ans =
    2    4
    6    8
```
See Also

num2str, str2double, hex2num, sscanf, sparse, char, special characters
**Purpose**
Concatenate strings horizontally

**Syntax**
t = strcat(s1, s2, s3, ...)

**Description**
t = strcat(s1, s2, s3, ...) horizontally concatenates corresponding rows of the character arrays s1, s2, s3, etc. All input arrays must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array.

When any of the inputs is a cell array of strings, strcat returns a cell array of strings formed by concatenating corresponding elements of s1, s2, etc. The inputs must all have the same size (or any can be a scalar). Any of the inputs can also be character arrays.

strcat ignores trailing ASCII white space characters and omits all such characters from the output. White space characters in ASCII are space, newline, carriage return, tab, vertical tab, or form-feed characters, all of which return a true response from the MATLABisspace function. Use the concatenation syntax [s1 s2 s3 ...] to preserve trailing spaces. strcat does not ignore inputs that are cell arrays of strings.

**Remarks**
strcat and matrix operation are different for strings that contain trailing spaces:

```matlab
a = 'hello ';
b = 'goodbye';
strcat(a, b)
ans =
hellogoodbye
```

```matlab
[a b]
ans =
hello goodbyee
```

**Examples**
Given two 1-by-2 cell arrays a and b,

```matlab
a =
'abcde'

b =
'fghi'

[ a b]
ans =
'abcdefghi'
```
the command \( t = \texttt{strcat}(a,b) \) yields

\[
\begin{align*}
\text{t} &= \text{'}abcdejk1\text{' } \text{'}fghimn\text{'}
\end{align*}
\]

Given the 1-by-1 cell array \( c = \{\texttt{`Q'}\} \), the command \( t = \texttt{strcat}(a,b,c) \) yields

\[
\begin{align*}
\text{t} &= \text{'}abcdejk1Q\text{' } \text{'}fghimnQ\text{'}
\end{align*}
\]

**See Also**

\texttt{strvcat}, \texttt{cat}, \texttt{vertcat}, \texttt{horzcat}, \texttt{cellstr}, \texttt{special character} [ ]
**Purpose**

Compare strings

**Syntax**

\[
\begin{align*}
TF &= \text{strcmp}('\text{str1}', '\text{str2}') \\
TF &= \text{strcmp}('\text{str}', C) \\
TF &= \text{strcmp}(C1, C2)
\end{align*}
\]

Each of these syntaxes applies to both strcmp and strcmpi. The strcmp function is case sensitive in matching strings, while strcmpi is not.

**Description**

Although the following descriptions show only strcmp, they apply to strcmpi as well. The two functions are the same except that strcmpi compares strings without sensitivity to letter case:

\[
\begin{align*}
TF &= \text{strcmp}('\text{str1}', '\text{str2}') \text{ compares the strings } \text{str1} \text{ and } \text{str2} \\
&\text{ and returns logical } 1 \text{ (true) if they are identical, and returns logical } 0 \text{ (false) otherwise. } \text{str1} \text{ and } \text{str2} \text{ can be character arrays of any dimension, but strcmp does not return true unless the sizes of both arrays are equal, and the contents of the two arrays are the same.}
\end{align*}
\]

\[
\begin{align*}
TF &= \text{strcmp}('\text{str}', C) \text{ compares string } \text{str} \text{ to the each element of } \\
&\text{ cell array } C, \text{ where } \text{str} \text{ is a character vector (or a 1-by-1 cell array) and } \\
&C \text{ is a cell array of strings. The function returns } TF, \text{ a logical array that is the same size as } C \text{ and contains logical } 1 \text{ (true) for those elements of } C \text{ that are a match, and logical } 0 \text{ (false) for those elements that are not. The order of the first two input arguments is not important.}
\end{align*}
\]

\[
\begin{align*}
TF &= \text{strcmp}(C1, C2) \text{ compares each element of } C1 \text{ to the same element in } C2, \text{ where } C1 \text{ and } C2 \text{ are equal-size cell arrays of strings. } \\
&\text{ Input } C1 \text{ or } C2 \text{ can also be a character array with the right number of rows. The function returns } TF, \text{ a logical array that is the same size as } C1 \text{ and } C2, \text{ and contains logical } 1 \text{ (true) for those elements of } C1 \text{ and } C2 \text{ that are a match, and logical } 0 \text{ (false) for those elements that are not.}
\end{align*}
\]

**Remarks**

These functions are intended for comparison of character data. When used to compare numeric data, they return logical 0.

Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
The value returned by `strcmp` and `strcmpi` is not the same as the C language convention.

`strcmp` and `strcmpi` support international character sets.

**Examples**

**Example 1**

Perform a simple comparison of two strings:

```matlab
cmp = 'Yes', 'No')
ans =
  0

strcmp('Yes', 'Yes')
ans =
  1
```

**Example 2**

Create 3 cell arrays of strings:

```matlab
A = {'MATLAB', 'SIMULINK'; ...
    'Toolboxes', 'The MathWorks'};

B = {'Handle Graphics', 'Real Time Workshop'; ...
    'Toolboxes', 'The MathWorks'};

C = {'handle graphics', 'Signal Processing'; ...
    'Toolboxes', 'The MATHWORKS'};
```

Compare cell arrays `A` and `B` with sensitivity to case:

```matlab
strcmp(A, B)
ans =
   0   0
   1   1
```

Compare cell arrays `B` and `C` without sensitivity to case. Note that `Toolboxes` doesn't match because of the leading space characters in `C{2,1}` that do not appear in `B{2,1}`:
Example 3

Compare a string vector to a cell array of strings, a string vector to a string array, and a string array to a cell array of strings. Start by creating a cell array of strings (cellArr), a string array containing the same strings plus space characters for padding (strArr), and a string vector containing one of the strings plus padding (strVec):

```matlab
cellArr = { ... 
    'There are 10 kinds of people in the world,'; ... 
    'those who understand binary math,'; ... 
    'and those who don''t.'};

strArr = char(cellArr);
strVec = strArr(2,:)
strVec = 
    those who understand binary math,
```

Remove the space padding from the string vector and compare it to the cell array. The MATLAB software compares the string with each row of the cell array, finding a match on the second row:

```matlab
strcmpi(B, C)
ans =
    1   0
    0   1
```

```matlab
strncpy(B, C)
ans =
    1   0
    0   1
```

```matlab
Example 3
```

```matlab
Compare a string vector to a cell array of strings, a string vector to a string array, and a string array to a cell array of strings. Start by creating a cell array of strings (cellArr), a string array containing the same strings plus space characters for padding (strArr), and a string vector containing one of the strings plus padding (strVec):

```matlab
cellArr = { ... 
    'There are 10 kinds of people in the world,'; ... 
    'those who understand binary math,'; ... 
    'and those who don''t.'};

strArr = char(cellArr);
strVec = strArr(2,:)
strVec = 
    those who understand binary math,
```

Remove the space padding from the string vector and compare it to the cell array. The MATLAB software compares the string with each row of the cell array, finding a match on the second row:

```matlab
strncpy(deblank(strVec), cellArr)
ans =
    0
    1
    0
```

Compare the string vector with the string array. Unlike the case above, MATLAB does not compare the string vector with each row of the string array. It compares the entire contents of one against the entire contents of the other:

```matlab
strncpy(strVec, strArr)
**strcmp, strcmpi**

```
ans =
  0
```

Lastly, compare each row of the three-row string array against the same rows of the cell array. MATLAB finds them all to be equivalent. Note that in this case you do not have to remove the space padding from the string array:

```
strcmp(strArr, cellArr)
ans =
  1
  1
  1
```

**See Also**

* strncmp, strncmpi, strmatch, strfind, findstr, regexp, regexpi, regexprep, regexptranslate*
Purpose
Compute 2-D streamline data

Syntax
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)

Description
XY = stream2(x,y,u,v,startx,starty) computes streamlines from vector data u and v. The arrays x and y define the coordinates for u and v and must be monotonic and 2-D plaid (such as the data produced by meshgrid). startx and starty define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value XY contains a cell array of vertex arrays.

XY = stream2(u,v,startx,starty) assumes the arrays x and y are defined as [x,y] = meshgrid(1:n,1:m) where [m,n] = size(u).

XY = stream2(...,options) specifies the options used when creating the streamlines. Define options as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

[stepsize]

or

[stepsize, max_number_vertices]

If you do not specify a value, MATLAB software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the streamline command to plot the data returned by stream2.

Examples
This example draws 2-D streamlines from data representing air currents over regions of North America.
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:,:,5),y(:,:,5),u(:,:,5),v(:,:,5),sx,sy));

See Also
coneplot, stream3, streamline
“Volume Visualization” on page 1-108 for related functions
Specifying Starting Points for Stream Plots for related information
Purpose
Compute 3-D streamline data

Syntax
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)
XYZ = stream3(U,V,W,startx,starty,startz)
XYZ = stream3(...,options)

Description
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz) computes streamlines from vector data U, V, W. The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (such as the data produced by meshgrid). startx, starty, and startz define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value XYZ contains a cell array of vertex arrays.

XYZ = stream3(U,V,W,startx,starty,startz) assumes the arrays X, Y, and Z are defined as [X,Y,Z] = meshgrid(1:N,1:M,1:P) where [M,N,P] = size(U).

XYZ = stream3(...,options) specifies the options used when creating the streamlines. Define options as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

[stepsize]

or

[stepsize, max_number_vertices]

If you do not specify values, MATLAB software uses the default:

• Step size = 0.1 (one tenth of a cell)
• Maximum number of vertices = 10000

Use the streamline command to plot the data returned by stream3.
Examples

This example draws 3-D streamlines from data representing air currents over regions of North America.

```matlab
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamline(stream3(x,y,z,u,v,w,sx,sy,sz))
view(3)
```

See Also

coneplot, stream2, streamline

“Volume Visualization” on page 1-108 for related functions

Specifying Starting Points for Stream Plots for related information
streamline

**Purpose**
Plot streamlines from 2-D or 3-D vector data

**GUI Alternatives**
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**
- `streamline(X,Y,Z,U,V,W,startx,starty,startz)`
- `streamline(U,V,W,startx,starty,startz)`
- `streamline(XYZ)`
- `streamline(X,Y,U,V,startx,starty)`
- `streamline(U,V,startx,starty)`
- `streamline(XY)`
- `streamline(...,options)`
- `h = streamline(...)`

**Description**
- `streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws streamlines from 3-D vector data U, V, W. The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). startx, starty, startz define the starting positions of the streamlines. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.
- `streamline(U,V,W,startx,starty,startz)` assumes the arrays X, Y, and Z are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P), where [M,N,P] = size(U).`
- `streamline(XYZ)` assumes XYZ is a precomputed cell array of vertex arrays (as produced by `stream3`).
- `streamline(X,Y,U,V,startx,starty)` draws streamlines from 2-D vector data U, V. The arrays X, Y define the coordinates for U, V and must
streamline

be monotonic and 2-D plaid (such as the data produced by meshgrid). startx and starty define the starting positions of the streamlines. The output argument h contains a vector of line handles, one handle for each streamline.

streamline(U,V,startx,starty) assumes the arrays X and Y are defined as [X,Y] = meshgrid(1:N,1:M), where [M,N] = size(U).

streamline(XY) assumes XY is a precomputed cell array of vertex arrays (as produced by stream2).

streamline(...,options) specifies the options used when creating the streamlines. Define options as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

    [stepsize]

or

    [stepsize, max_number_vertices]

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 1000

streamline(axes_handle,...) plots into the axes object with the handle axes_handle instead of the into current axes object (gca).

h = streamline(...) returns a vector of line handles, one handle for each streamline.

**Examples**

This example draws streamlines from data representing air currents over a region of North America. Loading the wind data set creates the variables x, y, z, u, v, and w in the MATLAB workspace.

The plane of streamlines indicates the flow of air from the west to the east (the x-direction) beginning at x = 80 (which is close to
the minimum value of the x coordinates). The y- and z-coordinate starting points are multivalued and approximately span the range of these coordinates. `meshgrid` generates the starting positions of the streamlines.

```matlab
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
h = streamline(x,y,z,u,v,w,sx,sy,sz);
set(h,'Color','red')
view(3)
```

See Also

`coneplot`, `stream2`, `stream3`, `streamparticles`

“Volume Visualization” on page 1-108 for related functions

Specifying Starting Points for Stream Plots for related information

Stream Line Plots of Vector Data for another example
**Purpose**
Plot stream particles

**GUI Alternatives**
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

```
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)
streamparticles(line_handle,...)
h = streamparticles(...)
```

**Description**

`streamparticles(vertices)` draws stream particles of a vector field. Stream particles are usually represented by markers and can show the position and velocity of a streamline. `vertices` is a cell array of 2-D or 3-D vertices (as if produced by `stream2` or `stream3`).

`streamparticles(vertices,n)` uses `n` to determine how many stream particles to draw. The `ParticleAlignment` property controls how `n` is interpreted.

- If `ParticleAlignment` is set to off (the default) and `n` is greater than 1, approximately `n` particles are drawn evenly spaced over the streamline vertices.

If `n` is less than or equal to 1, `n` is interpreted as a fraction of the original stream vertices; for example, if `n` is 0.2, approximately 20% of the vertices are used.

`n` determines the upper bound for the number of particles drawn. The actual number of particles can deviate from `n` by as much as a factor of 2.
• If ParticleAlignment is on, n determines the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value. The default value is \( n = 1 \).

\texttt{streamparticles(\ldots,'PropertyName',PropertyValue,\ldots)}
controls the stream particles using named properties and specified values. Any unspecified properties have default values. MATLAB ignores the case of property names.

**Stream Particle Properties**

Animate — Stream particle motion [nonnegative integer]

The number of times to animate the stream particles. The default is 0, which does not animate. \( \infty \) animates until you enter Ctrl+C.

FrameRate — Animation frames per second [nonnegative integer]

This property specifies the number of frames per second for the animation. \( \infty \), the default, draws the animation as fast as possible. Note that the speed of the animation might be limited by the speed of the computer. In such cases, the value of FrameRate cannot necessarily be achieved.

ParticleAlignment — Align particles with streamlines [ on | {off} ]

Set this property to on to draw particles at the beginning of each streamline. This property controls how \texttt{streamparticles} interprets the argument \( n \) (number of stream particles).

Stream particles are line objects. In addition to stream particle properties, you can specify any line object property, such as Marker and EraseMode. \texttt{streamparticles} sets the following line properties when called.

<table>
<thead>
<tr>
<th>Line Property</th>
<th>Value Set by \texttt{streamparticles}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EraseMode</td>
<td>xor</td>
</tr>
<tr>
<td>LineStyle</td>
<td>none</td>
</tr>
<tr>
<td>Marker</td>
<td>o</td>
</tr>
</tbody>
</table>
streamparticles

<table>
<thead>
<tr>
<th>Line Property</th>
<th>Value Set by streamparticles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MarkerEdgeColor</td>
<td>none</td>
</tr>
<tr>
<td>MarkerFaceColor</td>
<td>red</td>
</tr>
</tbody>
</table>

You can override any of these properties by specifying a property name and value as arguments to streamparticles. For example, this statement uses RGB values to set the MarkerFaceColor to medium gray:

```
streamparticles(vertices,'MarkerFaceColor',[.5 .5 .5])
```

```
streamparticles(line_handle,...) uses the line object identified by line_handle to draw the stream particles.
```

```
h = streamparticles(...) returns a vector of handles to the line objects it creates.
```

Examples

This example combines streamlines with stream particle animation. The interpstreamspeed function determines the vertices along the streamlines where stream particles will be drawn during the animation, thereby controlling the speed of the animation. Setting the axes DrawMode property to fast provides faster rendering.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
s1 = streamline(verts);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.025);
axis tight; view(30,30); daspect([1 1 .125])
camproj perspective; camva(8)
set(gca,'DrawMode','fast')
box on
streamparticles(iverts,35,'animate',10,'ParticleAlignment','on')
```

The following picture is a static view of the animation.
This example uses the streamlines in the $z = 5$ plane to animate the flow along these lines with streamparticles.

```matlab
load wind
daspect([1 1 1]); view(2)
[verts averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
sl = streamline([verts averts]);
axis tight off;
set(sl,'Visible','off')
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.05);
set(gca,'DrawMode','fast','Position',[0 0 1 1],'ZLim',[4.9 5.1])
set(gcf,'Color','black')
streamparticles(iverts, 200, ...
   'Animate',100,'FrameRate',40, ...
   'MarkerSize',10,'MarkerFaceColor','yellow')
```

See Also

interpstreamspeed, stream3, streamline

“Volume Visualization” on page 1-108 for related functions

Creating Stream Particle Animations for more details

Specifying Starting Points for Stream Plots for related information
streamribbon

**Purpose**
3-D stream ribbon plot from vector volume data

**GUI Alternatives**
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in *plot edit* mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

```
streamribbon(X,Y,Z,U,V,W,startx,starty,startz)
streamribbon(U,V,W,startx,starty,startz)
streamribbon(vertices,X,Y,Z,cav,speed)
streamribbon(vertices,cav,speed)
streamribbon(vertices,twistangle)
streamribbon(...,width)
streamribbon(axes_handle,...)
```

**Description**

`streamribbon(X,Y,Z,U,V,W,startx,starty,startz)` draws stream ribbons from vector volume data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `startx`, `starty`, and `startz` define the starting positions of the stream ribbons at the center of the ribbons. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

Generally, you should set the `DataAspectRatio` (`daspect`) before calling `streamribbon`.

`streamribbon(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression
streamribbon

\[ [X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p) \]

where \([m,n,p] = \text{size}(U)\).

\text{streamribbon}(\text{vertices},X,Y,Z,\text{cav},\text{speed})\) assumes precomputed streamline vertices, curl angular velocity, and flow speed. \text{vertices} is a cell array of streamline vertices (as produced by \text{stream3}). \(X, Y, Z, \text{cav},\) and \text{speed} are 3-D arrays.

\text{streamribbon}(\text{vertices},\text{cav},\text{speed})\) assumes \(X, Y,\) and \(Z\) are determined by the expression

\[ [X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p) \]

where \([m,n,p] = \text{size}(\text{cav})\).

\text{streamribbon}(\text{vertices},\text{twistangle})\) uses the cell array of vectors \text{twistangle} for the twist of the ribbons (in radians). The size of each corresponding element of \text{vertices} and \text{twistangle} must be equal.

\text{streamribbon}(\ldots,\text{width})\) sets the width of the ribbons to \text{width}.

\text{streamribbon}(\text{axes\_handle},\ldots)\) plots into the axes object with the handle \text{axes\_handle} instead of into the current axes object (\text{gca}).

\(h = \text{streamribbon}(\ldots)\) returns a vector of handles (one per start point) to surface objects.

**Examples**

This example uses stream ribbons to indicate the flow in the wind data set. Inputs include the coordinates, vector field components, and starting location for the stream ribbons.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
streamribbon(x,y,z,u,v,w,sx,dy,sz);
%-----Define viewing and lighting
axis tight
shading interp;
view(3);
```
This example uses precalculated vertex data (stream3), curl average velocity (curl1), and speed $\sqrt{u^2 + v^2 + w^2}$. Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to the previous example.
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*1;
streamribbon(verts,x,y,z,cav,spd);
%-----Define viewing and lighting
axis tight
shading interp
view(3)
camlight; lighting gouraud
This example specifies a twist angle for the stream ribbon.

```matlab
t = 0:.15:15;
verts = {cos(t)' sin(t)' (t/3)'};
twistangle = {cos(t)'};
daspect([1 1 1])
streamribbon(verts,twistangle);
%-----Define viewing and lighting
```
This example combines cone plots (coneplot) and stream ribbon plots in one graph.
streamribbon

%-----Define 3-D arrays x, y, z, u, v, w
xmin = -7; xmax = 7;
ymin = -7; ymax = 7;
zmin = -7; zmax = 7;
x = linspace(xmin,xmax,30);
y = linspace(ymin,ymax,20);
z = linspace(zmin,zmax,20);
[x y z] = meshgrid(x,y,z);
u = y; v = -x; w = 0*x+1;
daspect([1 1 1]);
[cx cy cz] = meshgrid(linspace(xmin,xmax,30),...
linspace(ymin,ymax,30),[-3 4]);

h = coneplot(x,y,z,u,v,w,cx,cy,cz,'quiver');
set(h,'color','k');

%-----Plot two sets of streamribbons
[sx sy sz] = meshgrid([-1 0 1],[-1 0 1],-6);
streamribbon(x,y,z,u,v,w,sx,dy,sz);

[sx sy sz] = meshgrid([1:6],[0],-6);
streamribbon(x,y,z,u,v,w,sx,dy,sz);

%-----Define viewing and lighting
shading interp
view(-30,10) ; axis off tight
camproj perspective; camva(66); camlookat;
camdolly(0,0,.5,'fixtarget')
camlight
See Also

curl, streamtube, streamline, stream3

“Volume Visualization” on page 1-108 for related functions

Displaying Curl with Stream Ribbons for another example

Specifying Starting Points for Stream Plots for related information
streamslice

Purpose
Plot streamlines in slice planes

GUI Alternatives
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax
streamslice(X,Y,Z,U,V,W,startx,starty,startz)
streamslice(U,V,W,startx,starty,startz)
streamslice(X,Y,U,V)
streamslice(U,V)
streamslice(...,density)
streamslice(...,'arrowsmode')
streamslice(...,'method')
streamslice(axes_handle,...)
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)

Description
streamslice(X,Y,Z,U,V,W,startx,starty,startz) draws well-spaced streamlines (with direction arrows) from vector data U, V, W in axis aligned x-, y-, z-planes starting at the points in the vectors startx, starty, startz. (The section Specifying Starting Points for Stream Plots provides more information on defining starting points.) The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (as if produced by meshgrid). U, V, W must be m-by-n-by-p volume arrays.

Do not assume that the flow is parallel to the slice plane. For example, in a stream slice at a constant z, the z component of the vector field W is ignored when you are calculating the streamlines for that plane.
Stream slices are useful for determining where to start streamlines, stream tubes, and stream ribbons. It is good practice to set the axes DataAspectRatio to \([1\ 1\ 1]\) when using `streamslice`.

`streamslice(U,V,W,startx,starty,startz)` assumes \(X, Y,\) and \(Z\) are determined by the expression

\[
[X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)
\]

where \([m,n,p] = \text{size}(U)\).

`streamslice(X,Y,U,V)` draws well-spaced streamlines (with direction arrows) from vector volume data \(U, V\). The arrays \(X, Y\) define the coordinates for \(U, V\) and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`streamslice(U,V)` assumes \(X, Y,\) and \(Z\) are determined by the expression

\[
[X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)
\]

where \([m,n,p] = \text{size}(U)\).

`streamslice(...,density)` modifies the automatic spacing of the streamlines. `density` must be greater than 0. The default value is 1; higher values produce more streamlines on each plane. For example, 2 produces approximately twice as many streamlines, while 0.5 produces approximately half as many.

`streamslice(...,'arrowsmode')` determines if direction arrows are present or not. `arrowsmode` can be

- `arrows` — Draw direction arrows on the streamlines (default).
- `noarrows` — Do not draw direction arrows.

`streamslice(...,'method')` specifies the interpolation method to use. `method` can be

- `linear` — Linear interpolation (default)
streamslice

- cubic — Cubic interpolation
- nearest — Nearest-neighbor interpolation

See interp3 for more information on interpolation methods.

streamslice(axes_handle,...) plots into the axes object with the handle axes_handle instead of into the current axes object (gca).

h = streamslice(...) returns a vector of handles to the line objects created.

[vertices arrowvertices] = streamslice(...) returns two cell arrays of vertices for drawing the streamlines and the arrows. You can pass these values to any of the streamline drawing functions (streamline, streamribbon, streamtube).

Examples

This example creates a stream slice in the wind data set at \( z = 5 \).

```matlab
load wind
daspect([1 1 1])
streamslice(x,y,z,u,v,w,[],[],[5])
axis tight
```
This example uses `streamslice` to calculate vertex data for the streamlines and the direction arrows. This data is then used by `streamline` to plot the lines and arrows. Slice planes illustrating with color the wind speed $\sqrt{u^2 + v^2 + w^2}$ are drawn by `slice` in the same planes.

```
load wind
```
daspect([1 1 1])
[verts averts] = streamslice(u,v,w,10,10,10);
streamline([verts averts])
spd = sqrt(u.^2 + v.^2 + w.^2);
hold on;
slice(spd,10,10,10);
colormap(hot)
shading interp
view(30,50); axis(volumebounds(spd));
camlight; material([.5 1 0])
This example superimposes contour lines on a surface and then uses \texttt{streamslice} to draw lines that indicate the gradient of the surface. \texttt{interp2} is used to find the points for the lines that lie on the surface.

\begin{verbatim}
  z = peaks;
surf(z)
shading interp
hold on
\end{verbatim}
streamslice

[c ch] = contour3(z,20); set(ch,'edgecolor','b')
[u v] = gradient(z);
h = streamslice(-u,-v);
set(h,'color','k')
for i=1:length(h);
    zi = interp2(z,get(h(i),'xdata'),get(h(i),'ydata'));
    set(h(i),'zdata',zi);
end
view(30,50); axis tight
See Also contourslice, slice, streamline, volumebounds

“Volume Visualization” on page 1-108 for related functions

Specifying Starting Points for Stream Plots for related information
streamtube

**Purpose**
Create 3-D stream tube plot

**GUI Alternatives**
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

**Syntax**

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
streamtube(axes_handle,...)
```

**Description**

`streamtube(X,Y,Z,U,V,W,startx,starty,startz)` draws stream tubes from vector volume data U, V, W. The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (as if produced by meshgrid). startx, starty, and startz define the starting positions of the streamlines at the center of the tubes. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The width of the tubes is proportional to the normalized divergence of the vector field.

Generally, you should set the `DataAspectRatio (daspect)` before calling `streamtube`.

`streamtube(U,V,W,startx,starty,startz)` assumes X, Y, and Z are determined by the expression
[X,Y,Z] = meshgrid(1:n,1:m,1:p)

where [m,n,p] = size(U).

streamtube(vertices,X,Y,Z,divergence) assumes precomputed streamline vertices and divergence. vertices is a cell array of streamline vertices (as produced by stream3). X, Y, Z, and divergence are 3-D arrays.

streamtube(vertices,divergence) assumes X, Y, and Z are determined by the expression

\[
[X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)
\]

where [m,n,p] = size(divergence).

streamtube(vertices,width) specifies the width of the tubes in the cell array of vectors, width. The size of each corresponding element of vertices and width must be equal. width can also be a scalar, specifying a single value for the width of all stream tubes.

streamtube(vertices) selects the width automatically.

streamtube(...,[scale n]) scales the width of the tubes by scale. The default is scale = 1. When the stream tubes are created, using start points or divergence, specifying scale = 0 suppresses automatic scaling. n is the number of points along the circumference of the tube. The default is n = 20.

streamtube(axes_handle,...) plots into the axes object with the handle axes_handle instead of into the current axes object (gca).

h = streamtube(...z) returns a vector of handles (one per start point) to surface objects used to draw the stream tubes.

Examples

This example uses stream tubes to indicate the flow in the wind data set. Inputs include the coordinates, vector field components, and starting location for the stream tubes.

load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamtube

daspect([1 1 1])
streamtube(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
view(3)
axis tight
shading interp;
camlight; lighting gouraud
This example uses precalculated vertex data (stream3) and divergence (divergence).

```matlab
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
streamtube(verts,x,y,z,-div);
%-----Define viewing and lighting
view(3)
axis tight
shading interp
camlight; lighting gouraud
```
streamtube

See Also
divergence, streamribbon, streamline, stream3
“Volume Visualization” on page 1-108 for related functions
Displaying Divergence with Stream Tubes for another example
Specifying Starting Points for Stream Plots for related information
Purpose
Find one string within another

Syntax
\[
\begin{align*}
  k &= \text{strfind}(\text{str}, \text{pattern}) \\
  k &= \text{strfind}([\text{cellstr}], \text{pattern})
\end{align*}
\]

Description
\( k = \text{strfind}(\text{str}, \text{pattern}) \) searches the string \( \text{str} \) for occurrences of
a shorter string, \( \text{pattern} \), and returns the starting index of each such
occurrence in the double array \( k \). If \( \text{pattern} \) is not found in \( \text{str} \), or if
\( \text{pattern} \) is longer than \( \text{str} \), then \text{strfind} returns the empty array \( [] \).

\( k = \text{strfind}(\text{cellstr}, \text{pattern}) \) searches each string in cell array
of strings \( \text{cellstr} \) for occurrences of a shorter string, \( \text{pattern} \), and
returns the starting index of each such occurrence in cell array \( k \). If
\( \text{pattern} \) is not found in a string or if \( \text{pattern} \) is longer then all strings
in the cell array, then \text{strfind} returns the empty array \( [] \), for that
string in the cell array.

The search performed by \text{strfind} is case sensitive. Any leading and
trailing blanks in \( \text{pattern} \) or in the strings being searched are explicitly
included in the comparison.

Examples
Use \text{strfind} to find a two-letter pattern in string \( S \):

\[
S = 'Find the starting indices of the pattern string';
\text{strfind}(S, 'in')
\text{ans} =
\begin{bmatrix}
  2 & 15 & 19 & 45
\end{bmatrix}
\]

\[
\text{strfind}(S, 'In')
\text{ans} =
\begin{bmatrix}
  []
\end{bmatrix}
\]

\[
\text{strfind}(S, ' ')
\text{ans} =
\begin{bmatrix}
  5 & 9 & 18 & 26 & 29 & 33 & 41
\end{bmatrix}
\]

Use \text{strfind} on a cell array of strings:
cstr = {'How much wood would a woodchuck chuck';
  'if a woodchuck could chuck wood?'};

idx = strfind(cstr, 'wood');

idx{:,1}
ans =
  10   23

ans =
    6   28

This means that 'wood' occurs at indices 10 and 23 in the first string
and at indices 6 and 28 in the second.

See Also
findstr, strmatch, strtok, strcmp, strncmp, strcmpi, strncmpi,
regexp, regexpi, regexprep
Purpose
String handling

Syntax

\[
S = 'Any Characters'
\]
\[
S = [S_1 S_2 ...]
\]
\[
S = strcat(S_1, S_2, ...)
\]

Description

\(S = 'Any Characters'\) creates a character array, or string. The string is actually a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The actual characters displayed depend on the character encoding scheme for a given font. The length of \(S\) is the number of characters. A quotation within the string is indicated by two quotes.

\(S = [S_1 S_2 ...]\) concatenates character arrays \(S_1, S_2, \text{ etc.}\) into a new character array, \(S\).

\(S = strcat(S_1, S_2, ...)\) concatenates \(S_1, S_2, \text{ etc.}\), which can be character arrays or “Cell Arrays of Strings”. When the inputs are all character arrays, the output is also a character array. When any of the inputs is a cell array of strings, \(strcat\) returns a cell array of strings.

Trailing spaces in \(strcat\) character array inputs are ignored and do not appear in the output. This is not true for \(strcat\) inputs that are cell arrays of strings. Use the \(S = [S_1 S_2 ...]\) concatenation syntax, shown above, to preserve trailing spaces.

\(S = char(X)\) can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.

\(X = double(S)\) converts the string to its equivalent double-precision numeric codes.

A collection of strings can be created in either of the following two ways:

- As the rows of a character array via \texttt{strvcat}
- As a cell array of strings via the curly braces

You can convert between character array and cell array of strings using \texttt{char} and \texttt{cellstr}. Most string functions support both types.
ischar(S) tells if S is a string variable. iscellstr(S) tells if S is a cell array of strings.

**Examples**

Create a simple string that includes a single quote.

```matlab
msg = 'You''re right!'
```

```matlab
msg =
You're right!
```

Create the string name using two methods of concatenation.

```matlab
name = ['Thomas' ' R. ' 'Lee']
name = strcat('Thomas',' R.',' Lee')
```

Create a vertical array of strings.

```matlab
C = strvcat('Hello','Yes','No','Goodbye')
```

```matlab
C =
Hello
Yes
No
Goodbye
```

Create a cell array of strings.

```matlab
S = {'Hello' 'Yes' 'No' 'Goodbye'}
```

```matlab
S =
'Hello' 'Yes' 'No' 'Goodbye'
```

**See Also**
char, isstrprop, cellstr, ischar, isletter, isspace, iscellstr, strvcat, sprintf, sscanf, text, input
**Purpose**

Justify character array

**Syntax**

```matlab
T = strjust(S)
T = strjust(S, 'right')
T = strjust(S, 'left')
T = strjust(S, 'center')
```

**Description**

*T = strjust(S)* or *T = strjust(S, 'right')* returns a right-justified version of the character array *S*.

*T = strjust(S, 'left')* returns a left-justified version of *S*.

*T = strjust(S, 'center')* returns a center-justified version of *S*.

**See Also**

debblank, strtrim
**Purpose**
Find possible matches for string

**Syntax**

```matlab
x = strmatch(str, strarray)
```

```matlab
x = strmatch(str, strarray, 'exact')
```

**Description**

`x = strmatch(str, strarray)` looks through the rows of the character array or cell array of strings `strarray` to find strings that begin with the text contained in `str`, and returns the matching row indices. Any trailing space characters in `str` or `strarray` are ignored when matching. `strmatch` is fastest when `strarray` is a character array.

`x = strmatch(str, strarray, 'exact')` compares `str` with each row of `strarray`, looking for an exact match of the entire strings. Any trailing space characters in `str` or `strarray` are ignored when matching.

**Examples**

The statement

```matlab
x = strmatch('max', strvcat('max', 'minimax', 'maximum'))
```

returns `x = [1; 3]` since rows 1 and 3 begin with 'max'. The statement

```matlab
x = strmatch('max', strvcat('max', 'minimax', 'maximum'),'exact')
```

returns `x = 1`, since only row 1 matches 'max' exactly.

**See Also**
`strcmp`, `strcmpi`, `strncmp`, `strncmpi`, `strfind`, `findstr`, `strvcat`, `regexp`, `regexpi`, `regexprep`
**Purpose**

Compare first n characters of strings

**Syntax**

\[ TF = \text{strncmp}('str1', 'str2', n) \]
\[ TF = \text{strncmp}('str', C, n) \]
\[ TF = \text{strncmp}(C1, C2, n) \]

Each of these syntaxes applies to both `strncmp` and `strncmpi`. The `strncmp` function is case sensitive in matching strings, while `strncmpi` is not.

**Description**

Although the following descriptions show only `strncmp`, they apply to `strncmpi` as well. The two functions are the same except that `strncmpi` compares strings without sensitivity to letter case:

\[ TF = \text{strncmp}('str1', 'str2', n) \] compares the first n characters of strings `str1` and `str2` and returns logical 1 (`true`) if they are identical, and returns logical 0 (`false`) otherwise. `str1` and `str2` can be character arrays of any dimension, but `strncmp` does not return `true` unless the sizes of both arrays are equal, and the contents of the two arrays are the same.

\[ TF = \text{strncmp}('str', C, n) \] compares the first n characters of `str` to the first n characters of each element of cell array `C`, where `str` is a character vector (or a 1-by-1 cell array), and `C` is a cell array of strings. The function returns `TF`, a logical array that is the same size as `C` and contains logical 1 (`true`) for those elements of `C` that are a match, and logical 0 (`false`) for those elements that are not. The order of the first two input arguments is not important.

\[ TF = \text{strncmp}(C1, C2, n) \] compares each element of `C1` to the same element in `C2`, where `C1` and `C2` are equal-size cell arrays of strings. Input `C1` or `C2` can also be a character array with the right number of rows. The function attempts to match only the first n characters of each string. The function returns `TF`, a logical array that is the same size as `C1` and `C2`, and contains logical 1 (`true`) for those elements of `C1` and `C2` that are a match, and logical 0 (`false`) for those elements that are not.
**Remarks**

These functions are intended for comparison of character data. When used to compare numeric data, they return logical 0.

Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

The value returned by `strncmp` and `strncmpi` is not the same as the C language convention.

`strncmp` and `strncmpi` support international character sets.

**Examples**

**Example 1**

From a list of 10 MATLAB functions, find those that apply to using a camera:

```matlab
function_list = {'calendar' 'case' 'camdolly' 'circshift' ...
    'caxis' 'camtarget' 'cast' 'camorbit' ...
    'callib' 'cart2sph'};

strncmp(function_list, 'cam', 3)
ans =
    0 0 1 0 0 1 0 1 0 0

function_list{strncmp(function_list, 'cam', 3)}
ans =
    camdolly
ans =
    camtarget
ans =
    camorbit
```

**Example 2**

Create two 5-by-10 string arrays `str1` and `str2` that are equal except for the element at row 4, column 3. Using linear indexing, this is element 14:

```matlab
str1 = ['AAAAAAAAAA'; 'BBBBBBBBBB'; 'CCCCCCCCCC'; ...
    'DDDDDDDDDD'; 'EEEEEEEEEE']
```
str1 =
    AAAAAAAAAA
    BBBB BBBB
    CCCCCCCC
    DDDDDD DDD
    EEEEEEEE

str2 = str1;
str2(4,3) = '-'
str2 =
    AAAAAAAAAA
    BBBB BBBB
    CCCCCCCC
    D -DDD DDD
    EEEEEEEE

Because MATLAB compares the arrays in linear order (that is, column by column rather than row by row), strncmp finds only the first 13 elements to be the same:

    str1   A B C D E A B C D E A B C D E
    str2   A B C D E A B C D E A B C - E
           | element 14

    strncmp(str1, str2, 13)
ans =
    1

    strncmp(str1, str2, 14)
ans =
    0

See Also
strcmp, strncmp, strmatch, strfind, findstr, regexp, regexpi, regexpprep, regexptranslate
strread

Purpose
Read formatted data from string

Note strread is not recommended. Use textscan to read data from a string.

Syntax
A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)

Description
A = strread('str') reads numeric data from input string str into a 1-by-N vector A, where N equals the number of whitespace-separated numbers in str. Use this form only with strings containing numeric data. See “Example 1” on page 2-3502 below.

[A, B, ...] = strread('str') reads numeric data from the string input str into scalar output variables A, B, and so on. The number of output variables must equal the number of whitespace-separated numbers in str. Use this form only with strings containing numeric data. See “Example 2” on page 2-3502 below.

[A, B, ...] = strread('str', 'format') reads data from str into variables A, B, and so on using the specified format. The number of output variables A, B, etc. must be equal to the number of format specifiers (e.g., %s or %d) in the format argument. You can read all of the data in str to a single output variable as long as you use only one format specifier in the command. See “Example 4” on page 2-3503 and “Example 5” on page 2-3503 below.

The table Formats for strread on page 2-3499 lists the valid format specifiers. More information on using formats is available under “Formats” on page 2-3501 in the Remarks section below.

[A, B, ...] = strread('str', 'format', N) reads data from str reusing the format string N times, where N is an integer greater than zero. If N is -1, strread reads the entire string. When str contains
only numeric data, you can set `format` to the empty string ("''"). See “Example 3” on page 2-3503 below.

```matlab
[A, B, ...] = strread('str', 'format', N, param, value, ...)
```
customizes `strread` using `param/value` pairs, as listed in the table `Parameters and Values for strread` on page 2-3500 below. When `str` contains only numeric data, you can set `format` to the empty string ("''"). The `N` argument is optional and may be omitted entirely. See “Example 7” on page 2-3504 below.

**Formats for strread**

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literals (ordinary characters)</td>
<td>Ignore the matching characters. For example, in a string that has <code>Dept</code> followed by a number (for department number), to skip the <code>Dept</code> and read only the number, use 'Dept' in the format string.</td>
<td>None</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>Read a signed integer value.</td>
<td>Double array</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>Read an integer value.</td>
<td>Double array</td>
</tr>
<tr>
<td><code>%f</code></td>
<td>Read a floating-point value.</td>
<td>Double array</td>
</tr>
<tr>
<td><code>%s</code></td>
<td>Read a white-space separated string.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td><code>%q</code></td>
<td>Read a double quoted string, ignoring the quotes.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>Read characters, including white space.</td>
<td>Character array</td>
</tr>
<tr>
<td><code>[% ...]</code></td>
<td>Read the longest string containing characters specified in the brackets.</td>
<td>Cell array of strings</td>
</tr>
</tbody>
</table>
## Formats for strread (Continued)

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>%[^...]</td>
<td>Read the longest nonempty string containing characters that are not specified in the brackets.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>%*...</td>
<td>Ignore the characters following *. See “Example 8” on page 2-3504 below.</td>
<td>No output</td>
</tr>
<tr>
<td>%w...</td>
<td>Read field width specified by w. The %f format supports %w.pf, where w is the field width and p is the precision.</td>
<td></td>
</tr>
</tbody>
</table>

## Parameters and Values for strread

<table>
<thead>
<tr>
<th>param</th>
<th>value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>whitespace</td>
<td>Any from the list below:</td>
<td>Treats vector of characters, *, as white space. Default is \b\r\n\t.</td>
</tr>
<tr>
<td></td>
<td>\b Backspace</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\n New line</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\r Carriage return</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\t Horizontal tab</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\ Backslash</td>
<td></td>
</tr>
<tr>
<td></td>
<td>% Percent sign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>' Single quotation mark</td>
<td></td>
</tr>
<tr>
<td>delimiter</td>
<td>Delimiter character</td>
<td>Specifies delimiter character. Default is one or more whitespace characters.</td>
</tr>
<tr>
<td>expchars</td>
<td>Exponent characters</td>
<td>Default is eEdD.</td>
</tr>
</tbody>
</table>
Parameters and Values for \texttt{strread} (Continued)

<table>
<thead>
<tr>
<th>param</th>
<th>value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufsize</td>
<td>Positive integer</td>
<td>Specifies the maximum string length, in bytes. Default is 4095.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>matlab</td>
<td>Ignores characters after %.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>shell</td>
<td>Ignores characters after #.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>c</td>
<td>Ignores characters between /* and */.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>c++</td>
<td>Ignores characters after //..</td>
</tr>
</tbody>
</table>

Remarks

Delimiters

If your data uses a character other than a space as a delimiter, you must use the \texttt{strread} parameter \texttt{delimiter} to specify the delimiter. For example, if the string \texttt{str} used a semicolon as a delimiter, you would use this command:

\[
\text{[names, types, x, y, answer] = strread(str,'\%s \%s \%f ... \%d \%s','delimiter',';')}
\]

Formats

The format string determines the number and types of return arguments. The number of return arguments must match the number of conversion specifiers in the format string.

The \texttt{strread} function continues reading \texttt{str} until the entire string is read. If there are fewer format specifiers than there are entities in \texttt{str}, \texttt{strread} reapplies the format specifiers, starting over at the beginning. See “Example 5” on page 2-3503 below.
The format string supports a subset of the conversion specifiers and conventions of the C language fscanf routine. White-space characters in the format string are ignored.

**Preserving White-Space**

If you want to preserve leading and trailing spaces in a string, use the whitespace parameter as shown here:

```matlab
str = ' An example of preserving spaces ';
strread(str, '%s', 'whitespace', '')
ans =
' An example of preserving spaces '
```

**Examples**

**Example 1**

Read numeric data into a 1-by-5 vector:

```matlab
a = strread('0.41 8.24 3.57 6.24 9.27')
a =
  0.4100   8.2400   3.5700   6.2400   9.2700
```

**Example 2**

Read numeric data into separate scalar variables:

```matlab
[a b c d e] = strread('0.41 8.24 3.57 6.24 9.27')
a =
  0.4100
b =
  8.2400
c =
  3.5700
d =
  6.2400
e =
  9.2700
```
Example 3
Read the only first three numbers in the string, also formatting as floating point:

\[
a = \text{strread}('0.41 8.24 3.57 6.24 9.27', '%4.2f', 3)
\]

\[
a =
0.4100
8.2400
3.5700
\]

Example 4
Truncate the data to one decimal digit by specifying format %3.1f. The second specifier, %*1d, tells strread not to read in the remaining decimal digit:

\[
a = \text{strread}('0.41 8.24 3.57 6.24 9.27', '%3.1f %*1d')
\]

\[
a =
0.4000
8.2000
3.5000
6.2000
9.2000
\]

Example 5
Read six numbers into two variables, reusing the format specifiers:

\[
[a \ b] = \text{strread}('0.41 8.24 3.57 6.24 9.27 3.29', '%f %f')
\]

\[
a =
0.4100
3.5700
9.2700
\]
\[
b =
8.2400
6.2400
\]
Example 6
Read string and numeric data to two output variables. Ignore commas in the input string:

```matlab
str = 'Section 4, Page 7, Line 26';

[name value] = strread(str, '%s %d,')
name =
  'Section'
  'Page'
  'Line'
value =
  4
  7
  26
```

Example 7
Read the string used in the last example, but this time delimiting with commas instead of spaces:

```matlab
str = 'Section 4, Page 7, Line 26';

[a b c] = strread(str, '%s %s %s', 'delimiter', ',')
a =
  'Section 4'
b =
  'Page 7'
c =
  'Line 26'
```

Example 8
Read selected portions of the input string:

```matlab
str = '<table border=5 width="100%" cellspacing=0>';

[border width space] = strread(str, ...
```
'%*s%*s %c %*s "%4s" %*s %c', 'delimiter', '=')
border =
      5
width =
  '100%'
space =
      0

Example 9
Read the string into two vectors, restricting the Answer values to T and F. Also note that two delimiters (comma and space) are used here:

str = 'Answer_1: T, Answer_2: F, Answer_3: F';

[a b] = strread(str, '%s %[TF]', 'delimiter', ',', '.')
a =
  'Answer_1:'
  'Answer_2:
  'Answer_3:'
b =
  'T'
  'F'
  'F'

See Also
    textscan, sscanf
strrep

Purpose
Find and replace substring

Syntax
str = strrep(str1, str2, str3)

Description
str = strrep(str1, str2, str3) replaces all occurrences of the string str2 within string str1 with the string str3.

strrep(str1, str2, str3), when any of str1, str2, or str3 is a cell array of strings, returns a cell array the same size as str1, str2, and str3 obtained by performing a strrep using corresponding elements of the inputs. The inputs must all be the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the right number of rows.

Examples
s1 = 'This is a good example.';
str = strrep(s1, 'good', 'great')
str =
This is a great example.
A =
'MATLAB' 'SIMULINK'
'Toolboxes' 'The MathWorks'
B =
'Handle Graphics' 'Real Time Workshop'
'Toolboxes' 'The MathWorks'
C =
'Signal Processing' 'Image Processing'
'MATLAB' 'SIMULINK'
strrep(A, B, C)
ans =
'MATLAB' 'SIMULINK'
'MATLAB' 'SIMULINK'

See Also
strfind
**Purpose**  
Selected parts of string

**Syntax**

token = strtok('str')  
token = strtok('str', delimiter)  
[token, remain] = strtok('str', ...)

**Description**

token = strtok('str') returns in token that part of the input string str that precedes the first white-space character (the default delimiter). Parsing of the string begins at the first nondelimiting (i.e., nonwhite-space) character and continues to the right until the MATLAB software either locates a delimiter or reaches the end of the string. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned.

White-space characters include space (ASCII 32), tab (ASCII 9), and carriage return (ASCII 13).

If str is a cell array of strings, token is a cell array of tokens.

token = strtok('str', delimiter) is the same as the above syntax except that you can specify one or more nondefault delimiters in the character vector, delimiter. Ignoring any leading delimiters, MATLAB returns in token that part of the input string that precedes one of the characters from the given delimiter vector.

[token, remain] = strtok('str', ...) returns in remain a substring of the input string that begins immediately after the token substring and ends with the last character in str. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned in token, and remain is an empty string ('').

If str is a cell array of strings, token is a cell array of tokens and remain is a character array.

**Examples**

**Example 1**

This example uses the default white-space delimiter:

```plaintext
s = ' This is a simple example.';
```
strtok

[token, remain] = strtok(s)
token = This
remain = is a simple example.

Example 2

Take a string of HTML code and break it down into segments delimited by the < and > characters. Write a while loop to parse the string and print each segment:

```c
s = sprintf('%s%s%s%s', ...
'\<ul class=continued><li class=continued>', ...
'\<pre><a name="13474"></a>token = strtok', ...
'\(''str''', delimiter)<a name="13475"></a>', ...
'token = strtok(''str'')');

remain = s;

while true
    [str, remain] = strtok(remain, '\<>');
    if isempty(str), break; end
    disp(sprintf('%s', str))
end
```

Here is the output:

```
ul class=continued
li class=continued
pre
a name="13474"
/a
token = strtok('str', delimiter)
a name="13475"
/a
token = strtok('str')
```
Example 3

Using strtok on a cell array of strings returns a cell array of strings in token and a character array in remain:

```matlab
s = {'all in good time'; ...
     'my dog has fleas'; ...
     'leave no stone unturned'};

remain = s;

for k = 1:4
    [token, remain] = strtok(remain);
    token
end
```

Here is the output:

```plaintext
token =
    'all'
    'my'
    'leave'
token =
    'in'
    'dog'
    'no'
token =
    'good'
    'has'
    'stone'
token =
    'time'
    'fleas'
    'unturned'
```

See Also

findstr, strmatch
**Purpose**
Remove leading and trailing white space from string

**Syntax**

S = strtrim(str)
C = strtrim(cstr)

**Description**

S = strtrim(str) returns a copy of string str with all leading and trailing white-space characters removed. A white-space character is one for which the isspace function returns logical 1 (true).

C = strtrim(cstr) returns a copy of the cell array of strings cstr with all leading and trailing white-space characters removed from each string in the cell array.

**Examples**

Remove the leading white-space characters (spaces and tabs) from str:

```matlab
str = sprintf(' 	 Remove leading white-space')
str =
    Remove      leading white-space
str = strtrim(str)
str =
    Remove      leading white-space
```

Remove leading and trailing white-space from the cell array of strings:

```matlab
cstr = {' Trim leading white-space';
         'Trim trailing white-space '};
cstr = strtrim(cstr)
cstr =
    'Trim leading white-space'
    'Trim trailing white-space'
```

**See Also**

isspace, cellstr, deblank, strjust
Purpose
Create structure array

Syntax
s = struct('field1', values1, 'field2', values2, ...)
s = struct('field1', {}, 'field2', {}, ...)
s = struct
s = struct([])
s = struct(obj)

Description
s = struct('field1', values1, 'field2', values2, ...) creates a structure array with the specified fields and values. Each value input (values1, values2, etc.), can either be a cell array or a scalar value. Those that are cell arrays must all have the same dimensions.

The size of the resulting structure is the same size as the value cell arrays, or 1-by-1 if none of the values is a cell array. Elements of the value array inputs are placed into corresponding structure array elements.

Note If any of the values fields is an empty cell array {}, the MATLAB software creates an empty structure array in which all fields are also empty.

Structure field names must begin with a letter, and are case-sensitive. The rest of the name may contain letters, numerals, and underscore characters. Use the namelengthmax function to determine the maximum length of a field name.

s = struct('field1', {}, 'field2', {}, ...) creates an empty structure with fields field1, field2, ...

s = struct creates a 1-by-1 structure with no fields.

s = struct([]) creates an empty structure with no fields.

s = struct(obj) creates a structure s that is identical to the underlying structure in the input object obj. MATLAB does not convert
struct

obj, but rather creates s as a new structure. This structure does not retain the class information in obj.

Remarks

Two Ways to Access Fields

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time.

Fields That Are Cell Arrays

To create fields that contain cell arrays, place the cell arrays within a value cell array. For instance, to create a 1-by-1 structure, type

```matlab
s = struct('strings',{'hello','yes'},'lengths',[5 3])
s=
    strings: {'hello'  'yes'}
    lengths: [5 3]
```

Specifying Cell Versus Noncell Values

When using the syntax

```matlab
s = struct('field1', values1, 'field2', values2, ...)
```

the values inputs can be cell arrays or scalar values. For those values that are specified as a cell array, MATLAB assigns each element of values{m,n,...} to the corresponding field in each element of structure s:

```matlab
s(m,n,...).fieldN = valuesN{m,n,...}
```

For those values that are scalar, MATLAB assigns that single value to the corresponding field for all elements of structure s:

```matlab
s(m,n,...).fieldN = valuesN
```

See Example 3, below.
### Examples

#### Example 1

The command

```plaintext
s = struct('type', {'big','little'}, 'color', {'red'}, ...
'x', {3 4})
```

produces a structure array `s`:

```plaintext
s =
1x2 struct array with fields:
type
color
x
```

The value arrays have been distributed among the fields of `s`:

```plaintext
s(1)
ans =
    type: 'big'
    color: 'red'
    x: 3

s(2)
ans =
    type: 'little'
    color: 'red'
    x: 4
```

#### Example 2

Similarly, the command

```plaintext
a.b = struct('z', {});
```

produces an empty structure `a.b` with field `z`.

```plaintext
a.b
ans =
    0x0 struct array with fields:
        z
```
### Example 3

This example initializes one field `f1` using a cell array, and the other `f2` using a scalar value:

```matlab
s = struct('f1', {1 3; 2 4}, 'f2', 25)
s =
2x2 struct array with fields:
  f1
  f2
```

Field `f1` in each element of `s` is assigned the corresponding value from the cell array `{1 3; 2 4}:

```matlab
s.f1
ans =
  1
ans =
  2
ans =
  3
ans =
  4
```

Field `f2` for all elements of `s` is assigned one common value because the values input for this field was specified as a scalar:

```matlab
s.f2
ans =
  25
ans =
  25
ans =
  25
ans =
  25
```
See Also

isstruct, fieldnames, isfield, orderfields, getfield, setfield, rmfield, substruct, deal, cell2struct, struct2cell, namelengthmax, dynamic field names
Purpose: Convert structure to cell array

Syntax: c = struct2cell(s)

Description: c = struct2cell(s) converts the m-by-n structure s (with p fields) into a p-by-m-by-n cell array c.
If structure s is multidimensional, cell array c has size [p size(s)].

Examples: The commands

```
clear s, s.category = 'tree';
s.height = 37.4; s.name = 'birch';
```

create the structure

```
s =
    category: 'tree'
    height: 37.4000
    name: 'birch'
```

Converting the structure to a cell array,

```
c = struct2cell(s)
```

```
c =
    'tree'
    [37.4000]
    'birch'
```

See Also: cell2struct, cell, iscell, struct, isstruct, fieldnames, dynamic field names
**Purpose**

Apply function to each field of scalar structure

**Syntax**

\[
A = \text{structfun}(\text{fun}, S) \\
[A, B, ...] = \text{structfun}(\text{fun}, S) \\
[A, ...] = \text{structfun}(\text{fun}, S, \text{'param1'}, \text{value1}, ...)
\]

**Description**

\(A = \text{structfun}(\text{fun}, S)\) applies the function specified by \(\text{fun}\) to each field of scalar structure \(S\), and returns the results in array \(A\). \(\text{fun}\) is a function handle to a function that takes one input argument and returns a scalar value. Return value \(A\) is a column vector that has one element for each field in input structure \(S\). The \(N\)th element of \(A\) is the result of applying \(\text{fun}\) to the \(N\)th field of \(S\), and the order of the fields is the same as that returned by a call to \text{fieldnames}. (\(A\) is returned as one or more scalar structures when the UniformOutput option is set to \text{false}. See the table below.)

\(\text{fun}\) must return values of the same class each time it is called. If \(\text{fun}\) is a handle to an overloaded function, then \text{structfun} follows MATLAB dispatching rules in calling the function.

\([A, B, ...] = \text{structfun}(\text{fun}, S)\) returns arrays \(A, B, ...\), each array corresponding to one of the output arguments of \(\text{fun}\). \text{structfun} calls \(\text{fun}\) each time with as many outputs as there are in the call to \text{structfun}. \(\text{fun}\) can return output arguments having different classes, but the class of each output must be the same each time \(\text{fun}\) is called.

\([A, ...] = \text{structfun}(\text{fun}, S, \text{'param1'}, \text{value1}, ...)\) enables you to specify optional parameter name/parameter value pairs. Parameters are
### Parameter | Value
--- | ---
'UniformOutput' | Logical value indicating whether or not the outputs of `fun` can be returned without encapsulation in a structure. The default value is true.

If equal to logical 1 (true), `fun` must return scalar values that can be concatenated into an array. The outputs can be any of the following types: numeric, logical, char, struct, or cell.

If equal to logical 0 (false), `structfun` returns a scalar structure or multiple scalar structures having fields that are the same as the fields of the input structure `S`. The values in the output structure fields are the results of calling `fun` on the corresponding values in the input structure `B`. In this case, the outputs can be of any data type.

'ErrorHandler' | Function handle specifying the function MATLAB is to call if the call to `fun` fails. MATLAB calls the error handling function with the following input arguments:

- A structure, with the fields 'identifier', 'message', and 'index', respectively containing the identifier of the error that occurred, the text of the error message, and the number of the field (in the same order as returned by `fieldnames`) at which the error occurred.

- The input argument at which the call to the function failed.

The error handling function should either rethrow an error or return the same number of outputs as `fun`. These outputs are then returned as the outputs of `structfun`. If 'UniformOutput' is true, the outputs of the error handler must also be scalars of the same type as the outputs of `fun`.

For example,

```matlab
function [A, B] = errorFunc(S, varargin)
    warning(S.identifier, S.message);
    A = NaN; B = NaN;
```
Examples

To create shortened weekday names from the full names, for example:
Create a structure with strings in several fields:

```matlab
s.f1 = 'Sunday';
s.f2 = 'Monday';
s.f3 = 'Tuesday';
s.f4 = 'Wednesday';
s.f5 = 'Thursday';
s.f6 = 'Friday';
s.f7 = 'Saturday';
```

```matlab
shortNames = structfun(@(x) ( x(1:3) ), s, ...
    'UniformOutput', false);
```

See Also

`cellfun`, `arrayfun`, `function_handle`, `cell2mat`, `spfun`
**strvcat**

**Purpose**
Concatenate strings vertically

**Syntax**

\[
S = \text{strvcat}(t_1, t_2, t_3, \ldots) \\
S = \text{strvcat}(c)
\]

**Description**
\[
S = \text{strvcat}(t_1, t_2, t_3, \ldots) \text{ forms the character array } S \\
\text{containing the text strings (or string matrices) } t_1, t_2, t_3, \ldots \text{ as rows.} \\
\text{Spaces are appended to each string as necessary to form a valid matrix.} \\
\text{Empty arguments are ignored.}
\]

\[
S = \text{strvcat}(c) \text{ when } c \text{ is a cell array of strings, passes each element of } c \text{ as an input to strvcat. Empty strings in the input are ignored.}
\]

**Remarks**
If each text parameter, \( t_i \), is itself a character array, \text{strvcat} appends them vertically to create arbitrarily large string matrices.

**Examples**
The command \text{strvcat('Hello','Yes')} is the same as \[
['Hello';'Yes ']
\], except that \text{strvcat} performs the padding automatically.

\[
t_1 = 'first'; t_2 = 'string'; t_3 = 'matrix'; t_4 = 'second';
\]

\[
S1 = \text{strvcat}(t_1, t_2, t_3)\quad S2 = \text{strvcat}(t_4, t_2, t_3)
\]

\[
S1 = \\
\text{first} \\
\text{string} \\
\text{matrix}
\]

\[
S2 = \\
\text{second} \\
\text{string} \\
\text{matrix}
\]

\[
S3 = \text{strvcat}(S1, S2)
\]

\[
S3 = \\
\text{first} \\
\text{string} \\
\text{matrix} \\
\text{second} \\
\text{string}
\]
matrix

See Also

strcat, cat, vertcat, horzcat, int2str, mat2str, num2str, strings, special character []
Purpose
Single index from subscripts

Syntax
IND = sub2ind(siz,I,J)
IND = sub2ind(siz,I1,I2,...,In)

Description
The sub2ind command determines the equivalent single index corresponding to a set of subscript values.

IND = sub2ind(siz,I,J) returns the linear index equivalent to the row and column subscripts I and J for a matrix of size siz. siz is a vector with ndim(A) elements (in this case, 2), where siz(1) is the number of rows and siz(2) is the number of columns.

IND = sub2ind(siz,I1,I2,...,In) returns the linear index equivalent to the n subscripts I1,I2,...,In for an array of size siz. siz is an n-element vector that specifies the size of each array dimension.

Examples
Create a 3-by-4-by-2 array, A.

A = [17 24 1 8; 2 22 7 14; 4 6 13 20];
A(:,:,2) = A - 10

A(:,:,1) =

17 24 1 8
2 22 7 14
4 6 13 20

A(:,:,2) =

7 14 -9 -2
-8 12 -3 4
-6 -4 3 10

The value at row 2, column 1, page 2 of the array is -8.

A(2,1,2)
ans =

-8

To convert \( A(2,1,2) \) into its equivalent single subscript, use `sub2ind`.

\[
\text{sub2ind(size(A),2,1,2)}
\]

\[
\text{ans =}
\]

\[
14
\]

You can now access the same location in \( A \) using the single subscripting method.

\[
A(14)
\]

\[
\text{ans =}
\]

\[
-8
\]

**See Also**

`ind2sub`, `find`, `size`
Purpose

Create axes in tiled positions

GUI

Alternatives

To add subplots to a figure, click one of the New Subplot icons in the Figure Palette, and slide right to select an arrangement of subplots. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation.

Syntax

- h = subplot(m,n,p) or subplot(mnp)
- subplot(m,n,p,'replace')
- subplot(m,n,P)
- subplot(h)
- subplot('Position',[left bottom width height])
- subplot(..., prop1, value1, prop2, value2, ...)
- h = subplot(...)
- subplot(m,n,p,'v6')

Description

subplot divides the current figure into rectangular panes that are numbered rowwise. Each pane contains an axes object. Subsequent plots are output to the current pane.

h = subplot(m,n,p) or subplot(mnp) breaks the figure window into an m-by-n matrix of small axes, selects the pth axes object for the current plot, and returns the axes handle. The axes are counted along the top row of the figure window, then the second row, etc. For example,

```
subplot(2,1,1), plot(income)
subplot(2,1,2), plot(outgo)
```
plots income on the top half of the window and outgoing on the bottom half. If the CurrentAxes is nested in a uipanel, the panel is used as the parent for the subplot instead of the current figure. The new axes object becomes the current axes.

`subplot(m,n,p,'replace')` If the specified axes object already exists, delete it and create a new axes.

`subplot(m,n,P)`, where `P` is a vector, specifies an axes position that covers all the subplot positions listed in `P`, including those spanned by `P`. For example, `subplot(2,3,[2 5])` creates one axes spanning positions 2 and 5 only (because there are no intervening locations in the grid), while `subplot(2,3,[2 6])` creates one axes spanning positions 2, 3, 5, and 6.

`subplot(h)` makes the axes object with handle `h` current for subsequent plotting commands.

`subplot('Position',[left bottom width height])` creates an axes at the position specified by a four-element vector. `left`, `bottom`, `width`, and `height` are in normalized coordinates in the range from 0.0 to 1.0.

`subplot(..., prop1, value1, prop2, value2, ...)` sets the specified property-value pairs on the subplot axis. To add the subplot to a specific figure pass the figure handle as the value for the `Parent` property. You cannot specify both a `Parent` and a `Position`; that is, `subplot('Position',[left bottom width height], 'Parent',h)` is not a valid syntax.

`h = subplot(...)` returns the handle to the new axes object.

**Backward-Compatible Version**

`subplot(m,n,p,'v6')` places the axes so that the plot boxes are aligned, but does not prevent the labels and ticks from overlapping. Saved subplots created with the `v6` option are compatible with MATLAB 6.5 and earlier versions.

Use the `subplot 'v6'` option and save the figure with the `v6` option when you want to be able to load a FIG-file containing subplots into MATLAB Version 6.5 or earlier.
Note  The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

Remarks

If a subplot specification causes a new axis to overlap a existing axis, the existing axis is deleted - unless the position of the new and existing axis are identical. For example, the statement subplot(1,2,1) deletes all existing axes overlapping the left side of the figure window and creates a new axis on that side—unless there is an axes there with a position that exactly matches the position of the new axes (and 'replace' was not specified), in which case all other overlapping axes will be deleted and the matching axes will become the current axes.

You can add subplots to GUIs as well as to figures. For information about creating subplots in a GUIDE-generated GUI, see “Creating Subplots” in the MATLAB Creating Graphical User Interfaces documentation.

If a subplot specification causes a new axes object to overlap any existing axes, subplot deletes the existing axes object and uicontrol objects. However, if the subplot specification exactly matches the position of an existing axes object, the matching axes object is not deleted and it becomes the current axes.

subplot(1,1,1) or clf deletes all axes objects and returns to the default subplot(1,1,1) configuration.

You can omit the parentheses and specify subplot as

    subplot mnp

where m refers to the row, n refers to the column, and p specifies the pane.
Be aware when creating subplots from scripts that the Position property of subplots is not finalized until either

- A drawnow command is issued.
- MATLAB returns to await a user command.

That is, the value obtained for subplot \( i \) by the command

\[
\text{get}(h(i), \text{'position'})
\]

will not be correct until the script refreshes the plot or exits.

**Special Case: subplot(111)**

The command `subplot(111)` is not identical in behavior to `subplot(1,1,1)` and exists only for compatibility with previous releases. This syntax does not immediately create an axes object, but instead sets up the figure so that the next graphics command executes a `clf reset` (deleting all figure children) and creates a new axes object in the default position. This syntax does not return a handle, so it is an error to specify a return argument. (MATLAB implements this behavior by setting the figure’s NextPlot property to `replace`.)

**Examples**

**Upper and Lower Subplots with Titles**

To plot income in the top half of a figure and outgo in the bottom half,

```matlab
income = [3.2 4.1 5.0 5.6];
outgo = [2.5 4.0 3.35 4.9];
subplot(2,1,1); plot(income)
title('Income')
subplot(2,1,2); plot(outgo)
title('Outgo')
```
Subplots in Quadrants

The following illustration shows four subplot regions and indicates the command used to create each.
**Assymetrical Subplots**

The following combinations produce asymmetrical arrangements of subplots.

```matlab
subplot(2,2,[1 3])
```
You can also use the colon operator to specify multiple locations if they are in sequence.
subplot(2,2,1:2)
subplot(2,2,3)
subplot(2,2,4)
Suppressing Axis Ticks

When you create many subplots in a figure, the axes tickmarks, which are shown by default, can either be obliterated or can cause axes to collapse, as the following code demonstrates:

```matlab
figure
for i=1:12
    subplot(12,1,i)
    plot (sin(1:100)*10^(i-1))
end
```
One way to get around this issue is to enlarge the figure to create enough space to properly display the tick labels.

Another approach is to eliminate the clutter by suppressing xticks and yticks for subplots as data are plotted into them. You can then label a single axes if the subplots are stacked, as follows:
figure
for i=1:12
    subplot(12,1,i)
    plot (sin(1:100)*10^(i-1))
    set(gca,'xtick',[],'ytick',[])
end
% Reset the bottom subplot to have xticks
set(gca,'xtickMode', 'auto')
See Also

axes, cla, clf, figure, gca

“Basic Plots and Graphs” on page 1-93 for more information
“Creating Subplots” in the MATLAB Creating Graphical User Interfaces documentation describes adding subplots to GUIs.
Purpose

Subscripted assignment for objects

Syntax

\[ A = \text{subsasgn}(A, S, B) \]

Description

\[ A = \text{subsasgn}(A, S, B) \]

is called for the syntax \( A(i)=B, A\{i\}=B, \) or \( A.i=B \) when \( A \) is an object. \( S \) is a structure array with the fields

- **type** — A string containing '()’, '{’, or '.’, where '()’ specifies integer subscripts, '{’ specifies cell array subscripts, and '.’ specifies subscripted structure fields.

- **subs** — A cell array or string containing the actual subscripts.

The MATLAB interpreter uses the built-in `subsasgn` function to interpret indexed assignment statements, such as \( A(i) = x; \).

You can modify the indexed assignment behavior of classes that you define (using the `classdef` syntax) by overloading `subsasgn` in your class. See “Indexed Reference and Assignment” for information on how to use `subsasgn` in your class definition.

Remarks

In the assignment \( A(J,K,\ldots) = B(M,N,\ldots) \), subscripts \( J, K, M, N, \) etc. can be scalar, vector, or arrays, provided that all of the following are true:

- The number of subscripts specified for \( B \), excluding trailing subscripts equal to 1, does not exceed the value returned by `ndims(B)`.

- The number of nonscalar subscripts specified for \( A \) equals the number of nonscalar subscripts specified for \( B \). For example, \( A(5,1:4,1,2) = B(5:8) \) is valid because both sides of the equation use one nonscalar subscript.

- The order and length of all nonscalar subscripts specified for \( A \) matches the order and length of nonscalar subscripts specified for \( B \). For example, \( A(1:4,\ 3,\ 3:9) = B(5:8,\ 1:7) \) is valid because both sides of the equation (ignoring the one scalar subscript 3) use a 4-element subscript followed by a 7-element subscript.
See the Remarks section of the `numel` reference page for information concerning the use of `numel` with regards to the overloaded `subsasgn` function.

**Classes Defined Prior to Version 7.6**

For MATLAB objects defined prior to Version 7.6, if `A` is an array of one of the “Fundamental MATLAB Classes”, then assigning a value to `A` with indexed assignment calls the built-in MATLAB `subsasgn` method. It does not call any `subsasgn` method that you might have overloaded for that class. For example, if `A` is an array of type `double`, and there is an `@double/subsasgn` method on your MATLAB path, the statement `A(I) = B` does not call this method, but calls the MATLAB built-in `subsasgn` method instead.

**Examples**

The syntax `A(1:2,:)=B` calls `A=subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs = {1:2,:}'.` A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}=B` calls `A=subsasgn(A,S,B)` where `S.type='{}'`.

The syntax `A.field=B` calls `subsasgn(A,S,B)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)=B` calls `A=subsasgn(A,S,B)` where `S` is a 3-by-1 structure array with the following values:

- `S(1).type='()'`  `S(2).type='.'`  `S(3).type='()'`
- `S(1).subs={1,2}`  `S(2).subs='name'`  `S(3).subs={3:5}`

**See Also**

`subsref`, `substruct`
Purpose  
Subscripted indexing using object as index

Syntax  
ind = subsindex(A)

Description  
ind = subsindex(A) is called for the syntax 'X(A)' when A is an object. subsindex must return the value of the object as a zero-based integer index. (ind must contain integer values in the range 0 to prod(size(X))-1.) subsindex is called by the default subsref and subsasgn functions, and you can call it if you overload these functions.

See “Indexing an Object with Another Object” for more information.

See Also  
subsasgn, subsref
Purpose
Angle between two subspaces

Syntax
theta = subspace(A,B)

Description
theta = subspace(A,B) finds the angle between two subspaces specified by the columns of A and B. If A and B are column vectors of unit length, this is the same as acos(A'*B).

Remarks
If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations A, and a second realization of the experiment described by B, subspace(A,B) gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples
Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

\[
H = \text{hadamard}(8);
A = H(:,2:4);
B = H(:,5:8);
\]

Note that matrices A and B are different sizes — A has three columns and B four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

\[
\text{theta} = \text{subspace}(A,B)
\]

\[
\text{theta} = 1.5708
\]

That A and B are orthogonal is shown by the fact that theta is equal to $\pi/2$.

\[
\text{theta} - \pi/2
\]

\[
\text{ans} = 0
\]
**Purpose**
Subscripted reference for objects

**Syntax**

\[
B = \text{subsref}(A, S)
\]

**Description**

\[
B = \text{subsref}(A, S)
\]
is called for the syntax \(A(i), A\{i\}, \text{or } A.i\) when \(A\) is an object. \(S\) is a structure array with the fields

- **type**: A string containing '( )', '{ }', or '.', where '( )' specifies integer subscripts, '{ }' specifies cell array subscripts, and '.' specifies subscripted structure fields.

- **subs**: A cell array or string containing the actual subscripts.

**Remarks**

subsref is designed to be used by the MATLAB interpreter to handle indexed references to objects. Calling subsref directly as a function is not recommended. If you do use subsref in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.

See the Remarks section of the numel reference page for information concerning the use of numel with regards to the overloaded subsref function.

If \(A\) is an array of one of the MATLAB built-in classes, then referencing a value of \(A\) using an indexed reference calls the built-in MATLAB subsref method. It does not call any subsref method that you may have overloaded for that class. For example, if \(A\) is an array of type double, and there is an @double/subsref method on your MATLAB path, the statement \(B = A(I)\) does not call this method, but calls the MATLAB built-in subsref method instead.

**Examples**

The syntax \(A(1:2,:)\) calls \(\text{subsref}(A,S)\) where \(S\) is a 1-by-1 structure with \(S\.\text{type}='( )'\) and \(S\.\text{subs}=[1:2,':']\). A colon used as a subscript is passed as the string ':'.

The syntax \(A\{1:2\}\) calls \(\text{subsref}(A,S)\) where \(S\.\text{type}='\{\}'\) and \(S\.\text{subs}=[1:2]\).

The syntax \(A\.\text{field}\) calls \(\text{subsref}(A,S)\) where \(S\.\text{type}='.'\) and \(S\.\text{subs}='\text{field}'\).
These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is a 3-by-1 structure array with the following values:

```matlab
S(1).type='()'  S(2).type='.'  S(3).type='()
S(1).subs={1,2}  S(2).subs='name'  S(3).subs={3:5}
```

### See Also

`numel`, `subsasgn`, `substruct`

See “Indexed Reference and Assignment” for information on object indexing and `subsref`. 
**Purpose**
Create structure argument for subsasgn or subsref

**Syntax**

\[ S = \text{substruct}(\text{type1}, \text{subs1}, \text{type2}, \text{subs2}, ...) \]

**Description**

\[ S = \text{substruct}(\text{type1}, \text{subs1}, \text{type2}, \text{subs2}, ...) \]
creates a structure with the fields required by an overloaded subsref or subsasgn method. Each type string must be one of '. ', '()', or '{}'. The corresponding subs argument must be either a field name (for the '.' type) or a cell array containing the index vectors (for the '(' or '{}' types).

The output S is a structure array containing the fields

- **type**: one of '.', '()', or '{}'
- **subs**: subscript values (field name or cell array of index vectors)

**Examples**

To call subsref with parameters equivalent to the syntax

\[ B = A(3,5).\text{field} \]

you can use

\[ S = \text{substruct}('()', \{3,5\}, '.', 'field'); \]
\[ B = \text{subsref}(A, S); \]

The structure created by substruct in this example contains the following:

\[ S(1) \]
\[ \text{ans} = \]
\[ \begin{array}{l}
\text{type: '()'} \\
\text{subs: {\{3\} \{5\}}} \\
\end{array} \]

\[ S(2) \]
ans =

    type: '.'
    subs: 'field'

See Also

subsasgn, subsref
Purpose

Extract subset of volume data set

Syntax

\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(X,Y,Z,V,\text{limits})
\]
\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(V,\text{limits})
\]
\[
Nv = \text{subvolume}(\ldots)
\]

Description

\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(X,Y,Z,V,\text{limits})
\]
extracts a subset of the volume data set \(V\) using the specified axis-aligned \(\text{limits}\). \(\text{limits} = [\text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}, \text{zmin}, \text{zmax}]\) (Any NaNs in the limits indicate that the volume should not be cropped along that axis.)

The arrays \(X\), \(Y\), and \(Z\) define the coordinates for the volume \(V\). The subvolume is returned in \(Nv\) and the coordinates of the subvolume are given in \(NX\), \(NY\), and \(NZ\).

\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(V,\text{limits})
\]
assumes the arrays \(X\), \(Y\), and \(Z\) are defined as

\[
[X,Y,Z] = \text{meshgrid}(1:N,1:M,1:P)
\]

where \([M,N,P] = \text{size}(V)\). \(Nv = \text{subvolume}(\ldots)\) returns only the subvolume.

Examples

This example uses a data set that is a collection of MRI slices of a human skull. The data is processed in a variety of ways:

- The 4-D array is squeezed (squeeze) into three dimensions and then a subset of the data is extracted (subvolume).
- The outline of the skull is an isosurface generated as a patch (p1) whose vertex normals are recalculated to improve the appearance when lighting is applied (patch, isosurface, isonormals).
- A second patch (p2) with interpolated face color draws the end caps (FaceColor, isocaps).
- The view of the object is set (view, axis, daspect).
- A 100-element grayscale colormap provides coloring for the end caps (colormap).

- Adding lights to the right and left of the camera illuminates the object (camlight, lighting).

```matlab
load mri
D = squeeze(D);
[x,y,z,D] = subvolume(D,[60,80,nan,80,nan,nan]);
p1 = patch(isosurface(x,y,z,D, 5),...
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
    'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight right; camlight left; lighting gouraud
```
See Also  
isocaps, isonormals, isosurface, reducepatch, reducevolume, smooth3

“Volume Visualization” on page 1-108 for related functions
**Purpose**

Sum of array elements

**Syntax**

\[
\begin{align*}
B &= \text{sum}(A) \\
B &= \text{sum}(A, \text{dim}) \\
B &= \text{sum}(\ldots, \text{'double'}) \\
B &= \text{sum}(\ldots, \text{dim}, \text{'double'}) \\
B &= \text{sum}(\ldots, \text{'native'}) \\
B &= \text{sum}(\ldots, \text{dim}, \text{'native'})
\end{align*}
\]

**Description**

\(B = \text{sum}(A)\) returns sums along different dimensions of an array.

If \(A\) is a vector, \(\text{sum}(A)\) returns the sum of the elements.

If \(A\) is a matrix, \(\text{sum}(A)\) treats the columns of \(A\) as vectors, returning a row vector of the sums of each column.

If \(A\) is a multidimensional array, \(\text{sum}(A)\) treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

\(B = \text{sum}(A, \text{dim})\) sums along the dimension of \(A\) specified by scalar \(\text{dim}\).

The \(\text{dim}\) input is an integer value from 1 to \(N\), where \(N\) is the number of dimensions in \(A\). Set \(\text{dim}\) to 1 to compute the sum of each column, 2 to sum rows, etc.

\(B = \text{sum}(\ldots, \text{'double'})\) and \(B = \text{sum}(\ldots, \text{dim}, \text{'double'})\) performs additions in double-precision and return an answer of type \text{double}, even if \(A\) has data type \text{single} or an integer data type. This is the default for integer data types.

\(B = \text{sum}(\ldots, \text{'native'})\) and \(B = \text{sum}(\ldots, \text{dim}, \text{'native'})\) performs additions in the native data type of \(A\) and return an answer of the same data type. This is the default for \text{single} and \text{double}.

**Remarks**

\(\text{sum(diag(X))}\) is the trace of \(X\).

**Examples**

The magic square of order 3 is

\[
\begin{align*}
M &= \text{magic}(3) \\
M &=
\end{align*}
\]
8  1  6
3  5  7
4  9  2

This is called a magic square because the sums of the elements in each column are the same.

\[
\text{sum}(M) = \\
15 \quad 15 \quad 15
\]

as are the sums of the elements in each row, obtained either by transposing or using the \text{dim} argument.

- Transposing

\[
\text{sum}(M') = \\
15 \quad 15 \quad 15
\]

- Using the \text{dim} argument

\[
\text{sum}(M,1) \\
\text{ans} = \\
15 \quad 15 \quad 15
\]

\section*{Nondouble Data Type Support}

This section describes the support of \text{sum} for data types other than \text{double}.

\subsection*{Data Type single}

You can apply \text{sum} to an array of type \text{single} and MATLAB software returns an answer of type \text{single}. For example,

\[
\text{sum}(	ext{single}([2 5 8]))
\]

\[
\text{ans} = \\
15
\]
class(ans)
ans =
single

**Integer Data Types**

When you apply `sum` to any of the following integer data types, MATLAB software returns an answer of type `double`:

- `int8` and `uint8`
- `int16` and `uint16`
- `int32` and `uint32`

For example,

```MATLAB
sum(single([2 5 8]));
class(ans)
```

ans =
single

If you want MATLAB to perform additions on an integer data type in the same integer type as the input, use the syntax

```MATLAB
sum(int8([2 5 8], 'native');
class(ans)
```

ans =
int8

**See Also**

`accumarray`, `cumsum`, `diff`, `isfloat`, `prod`
Purpose

Sum of timeseries data

Syntax

ts_sm = sum(ts)
ts_sm = sum(ts,'PropertyName1',PropertyValue1,...)

Description

ts_sm = sum(ts) returns the sum of the time-series data. When ts.Data is a vector, ts_sm is the sum of ts.Data values. When ts.Data is a matrix, ts_sm is a row vector containing the sum of each column of ts.Data (when IsTimeFirst is true and the first dimension of ts is aligned with time). For the N-dimensional ts.Data array, sum always operates along the first nonsingleton dimension of ts.Data.

ts_sm = sum(ts,'PropertyName1',PropertyValue1,...) specifies the following optional input arguments:

- 'MissingData' property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- 'Quality' values are specified by a vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- 'Weighting' property has two possible values, 'none' (default) or 'time'. When you specify 'time', larger time values correspond to larger weights.

Examples

1 Load a 24-by-3 data array.

   load count.dat

2 Create a timeseries object with 24 time values.

   count_ts = timeseries(count,1:24,'Name','CountPerSecond')

3 Calculate the sum of each data column for this timeseries object.

   sum(count_ts)
```
ans =
   768    1117    1574
```

The sum is calculated independently for each data column in the `timeseries` object.

**See Also**

`iqr (timeseries)`, `mean (timeseries)`, `median (timeseries)`, `std (timeseries)`, `var (timeseries)`, `timeseries`
Purpose Establish superior class relationship

Syntax superiorto('class1', 'class2', ...)

Description The superiorto function establishes a precedence that determines which object method is called.

Note You can use this function only from a constructor that calls the class function to create an object, which was the only way to create MATLAB classes prior to MATLAB Version 7.6.

See Object-Oriented Programming for information on the creating MATLAB classes.

superiorto('class1', 'class2', ...) invoked within a class constructor method, establishes that class as having precedence over the classes in the function argument list for purposes of function dispatching (i.e., which method or function is called in any given situation).

Remarks Suppose a is an object of class 'class_a', b is an object of class 'class_b' and c is an object of class 'class_c'. Also suppose the constructor method for class_c.m contains the statement superiorto('class_a'). Then, either of the following two statements:

   e = fun(a,c);
   e = fun(c,a);

invokes class_c/fun.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the left-most object’s method is called. So fun(b,c) calls class_b/fun, while fun(c,b) calls class_c/fun.

See Also inferiorto
Purpose
Open MathWorks Technical Support Web page

Syntax
support

Description

This Web page contains resources including

- A search engine, including an option for solutions to common problems
- Information about installation and licensing
- A patch archive for bug fixes you can download
- Other useful resources

See Also
doc, web
Purpose

3-D shaded surface plot

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

surf(Z)
surf(Z,C)
surf(X,Y,Z)
surf(X,Y,Z,C)
surf(...,'PropertyName',PropertyValue)
surf(axes_handles,...)
surfc(...)
h = surf(...)
hsurface = surf('v6',...)
hsurface = surfc('v6',...)

Description

Use surf and surfc to view mathematical functions over a rectangular region. surf and surfc create colored parametric surfaces specified by X, Y, and Z, with color specified by Z or C.

surf(Z) creates a three-dimensional shaded surface from the z components in matrix Z, using \( x = 1:n \) and \( y = 1:m \), where \([m,n] = \text{size}(Z)\). The height, Z, is a single-valued function defined over a geometrically rectangular grid. Z specifies the color data as well as surface height, so color is proportional to surface height.

surf(Z,C) plots the height of Z, a single-valued function defined over a geometrically rectangular grid, and uses matrix C, assumed to be the same size as Z, to color the surface.
surf(X,Y,Z) creates a shaded surface using Z for the color data as well as surface height. X and Y are vectors or matrices defining the x and y components of a surface. If X and Y are vectors, length(X) = n and length(Y) = m, where [m,n] = size(Z). In this case, the vertices of the surface faces are (X(j), Y(i), Z(i,j)) triples. To create X and Y matrices for arbitrary domains, use the meshgrid function.

surf(X,Y,Z,C) creates a shaded surface, with color defined by C. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

surf(...,'PropertyName',PropertyValue) specifies surface properties along with the data.

surf(axes_handles,...) and surfc(axes_handles,...) plot into the axes with handle axes_handle instead of the current axes (gca).

surfc(...) draws a contour plot beneath the surface.

h = surf(...) and h = surfc(...) return a handle to a surfaceplot graphics object.

**Backward-Compatible Version**

hsurface = surf('v6',...) and hsurface = surfc('v6',...) return the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

**Note** The v6 option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB.

See Plot Objects and Backward Compatibility for more information.

**Remarks**
surf and surfc do not accept complex inputs.

**Algorithm**
Abstractly, a parametric surface is parameterized by two independent variables, i and j, which vary continuously over a rectangle; for
example, $1 \leq i \leq m$ and $1 \leq j \leq n$. The three functions $x(i, j)$, $y(i, j)$, and $z(i, j)$ specify the surface. When $i$ and $j$ are integer values, they define a rectangular grid with integer grid points. The functions $x(i, j)$, $y(i, j)$, and $z(i, j)$ become three $m$-by-$n$ matrices, $X$, $Y$, and $Z$. Surface color is a fourth function, $c(i, j)$, denoted by matrix $C$.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

\[
\begin{array}{c}
  i-1, j \\
  \mid \\
  i,j-1 - i,j - i,j+1 \\
  \mid \\
  i+1,j
\end{array}
\]

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, $[X(:) \ Y(:) \ Z(:)]$ returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

Surface color can be specified in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of $x$ and $y$. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The shading function sets the shading. If the shading is *interp*, $C$ must be the same size as $X$, $Y$, and $Z$; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is *faceted* (the default) or *flat*, $C(i, j)$ specifies the constant color in the surface patch:

\[
\begin{array}{c}
  (i,j) - (i,j+1) \\
  \mid \text{ } C(i,j) \text{ } \mid \\
  (i+1,j) - (i+1,j+1)
\end{array}
\]
In this case, \( C \) can be the same size as \( X \), \( Y \), and \( Z \) and its last row and column are ignored. Alternatively, its row and column dimensions can be one less than those of \( X \), \( Y \), and \( Z \).

The `surf` and `surfc` functions specify the viewpoint using `view(3)`.

The range of \( X \), \( Y \), and \( Z \) or the current setting of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties (also set by the `axis` function) determines the axis labels.

The range of \( C \) or the current setting of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function) determines the color scaling. The scaled color values are used as indices into the current colormap.

**Examples**

Display a surface plot and contour plot of the `peaks` surface.

```matlab
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([-3 3 -3 3 -10 5])
```
Color a sphere with the pattern of +1s and -1s in a Hadamard matrix.

```matlab
k = 5;
n = 2^k-1;
[x,y,z] = sphere(n);
c = hadamard(2^k);
surf(x,y,z,c);
colormap([1 1 0; 0 1 1])
axis equal
```
surf, surfc

See Also
axis, caxis, colormap, contour, delaunay, imagesc, mesh, meshgrid, pcolor, shading, trisurf, view

Properties for surfaceplot graphics objects

“Surface and Mesh Creation” on page 1-104 for related functions

“Creating Mesh and Surface Plots” in the Getting Started with MATLAB documentation for background and examples.

Representing a Matrix as a Surface in the MATLAB 3-D Visualization documentation for further examples

Coloring Mesh and Surface Plots for information about how to control the coloring of surfaces
**Purpose**

Convert surface data to patch data

**Syntax**

\[
\text{fvc} = \text{surf2patch}(Z) \\
\text{fvc} = \text{surf2patch}(Z,C) \\
\text{fvc} = \text{surf2patch}(X,Y,Z) \\
\text{fvc} = \text{surf2patch}(X,Y,Z,C) \\
\text{fvc} = \text{surf2patch}(..., \text{'triangles'}) \\
[f,v,c] = \text{surf2patch}(...) \\
\]

**Description**

\( \text{fvc} = \text{surf2patch}(h) \)

converts the geometry and color data from the \texttt{surface} object identified by the handle \( h \) into patch format and returns the face, vertex, and color data in the struct \( \text{fvc} \). You can pass this struct directly to the \texttt{patch} command.

\( \text{fvc} = \text{surf2patch}(Z) \) calculates the patch data from the surface’s \( \text{ZData} \) matrix \( Z \).

\( \text{fvc} = \text{surf2patch}(Z,C) \) calculates the patch data from the surface’s \( \text{ZData} \) and \( \text{CData} \) matrices \( Z \) and \( C \).

\( \text{fvc} = \text{surf2patch}(X,Y,Z) \) calculates the patch data from the surface’s \( \text{XData} \), \( \text{YData} \), and \( \text{ZData} \) matrices \( X \), \( Y \), and \( Z \).

\( \text{fvc} = \text{surf2patch}(X,Y,Z,C) \) calculates the patch data from the surface’s \( \text{XData} \), \( \text{YData} \), \( \text{ZData} \), and \( \text{CData} \) matrices \( X \), \( Y \), \( Z \), and \( C \).

\( \text{fvc} = \text{surf2patch}(..., \text{'triangles'}) \) creates triangular faces instead of the quadrilaterals that compose surfaces.

\( [f,v,c] = \text{surf2patch}(...) \) returns the face, vertex, and color data in the three arrays \( f \), \( v \), and \( c \) instead of a struct.

**Examples**

The first example uses the \texttt{sphere} command to generate the \( \text{XData} \), \( \text{YData} \), and \( \text{ZData} \) of a surface, which is then converted to a patch. Note that the \( \text{ZData} \) (\( z \)) is passed to \texttt{surf2patch} as both the third and fourth arguments — the third argument is the \( \text{ZData} \) and the fourth argument is taken as the \( \text{CData} \). This is because the \texttt{patch} command does not
surf2patch

automatically use the \( z \)-coordinate data for the color data, as does the \texttt{surface} command.

Also, because \texttt{patch} is a low-level command, you must set the \texttt{view} to 3-D and shading to \texttt{faceted} to produce the same results produced by the \texttt{surf} command.

\[
[x\ y\ z] = \text{sphere};
\text{patch(surf2patch(x,y,z,z))};
\text{shading faceted; view(3)}
\]

In the second example \texttt{surf2patch} calculates face, vertex, and color data from a surface whose handle has been passed as an argument.

\[
s = \text{surf(peaks)};
\text{pause}
\text{patch(surf2patch(s))};
\text{delete(s)}
\text{shading faceted; view(3)}
\]

\textbf{See Also}\n
\texttt{patch, reducepatch, shrinkfaces, surface, surf}

“Volume Visualization” on page 1-108 for related functions
Purpose

Create surface object

Syntax

\[
\text{surface}(Z) \\
\text{surface}(Z,C) \\
\text{surface}(X,Y,Z) \\
\text{surface}(X,Y,Z,C) \\
\text{surface}(x,y,Z) \\
\text{surface}(\ldots'\text{PropertyName}',\text{PropertyValue},\ldots) \\
h = \text{surface}(\ldots)
\]

Description

\text{surface} is the low-level function for creating surface graphics objects. Surfaces are plots of matrix data created using the row and column indices of each element as the \(x\)- and \(y\)-coordinates and the value of each element as the \(z\)-coordinate.

\text{surface}(Z) plots the surface specified by the matrix \(Z\). Here, \(Z\) is a single-valued function, defined over a geometrically rectangular grid.

\text{surface}(Z,C) plots the surface specified by \(Z\) and colors it according to the data in \(C\) (see "Examples").

\text{surface}(X,Y,Z) uses \(C = Z\), so color is proportional to surface height above the \(x\)-\(y\) plane.

\text{surface}(X,Y,Z,C) plots the parametric surface specified by \(X\), \(Y\), and \(Z\), with color specified by \(C\).

\text{surface}(x,y,Z), \text{surface}(x,y,Z,C) replaces the first two matrix arguments with vectors and must have \text{length}(x) = n and \text{length}(y) = m where \([m,n] = \text{size}(Z)\). In this case, the vertices of the surface facets are the triples \((x(j),y(i),Z(i,j))\). Note that \(x\) corresponds to the columns of \(Z\) and \(y\) corresponds to the rows of \(Z\). For a complete discussion of parametric surfaces, see the \text{surf} function.

\text{surface}(\ldots'\text{PropertyName}',\text{PropertyValue},\ldots) follows the \(X\), \(Y\), \(Z\), and \(C\) arguments with property name/property value pairs to specify additional surface properties.

\(h = \text{surface}(\ldots)\) returns a handle to the created surface object.
Remarks

**Remarks**

**surface** does not respect the settings of the figure and axes **NextPlot** properties. It simply adds the surface object to the current axes.

If you do not specify separate color data (C), MATLAB uses the matrix (Z) to determine the coloring of the surface. In this case, color is proportional to values of Z. You can specify a separate matrix to color the surface independently of the data defining the area of the surface.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see **set** and **get** for examples of how to specify these data types).

**surface** provides convenience forms that allow you to omit the property name for the **XData**, **YData**, **ZData**, and **CData** properties. For example,

```matlab
surface(‘XData’,X,’YData’,Y,’ZData’,Z,’CData’,C)
```

is equivalent to

```matlab
surface(X,Y,Z,C)
```

When you specify only a single matrix input argument,

```matlab
surface(Z)
```

MATLAB assigns the data properties as if you specified

```matlab
surface(‘XData’,[1:size(Z,2)],...
     ‘YData’,[1:size(Z,1)],...
     ‘ZData’,Z,...
     ‘CData’,Z)
```

The **axis**, **caxis**, **colormap**, **hold**, **shading**, and **view** commands set graphics properties that affect surfaces. You can also set and query surface property values after creating them using the **set** and **get** commands.

Example

**Example**

This example creates a surface using the **peaks** M-file to generate the data, and colors it using the clown image. The **ZData** is a 49-by-49
element matrix, while the CData is a 200-by-320 matrix. You must set the surface’s FaceColor to texturemap to use ZData and CData of different dimensions.

load clown
surface(peaks,flipud(X),...
    'FaceColor','texturemap',...
    'EdgeColor','none',...
    'CDataMapping','direct')
colormap(map)
view(-35,45)
Note the use of the `surface(Z, C)` convenience form combined with property name/property value pairs.

Since the clown data (X) is typically viewed with the `image` command, which MATLAB normally displays with 'ij' axis numbering and direct `CDataMapping`, this example reverses the data in the vertical direction using `flipud` and sets the `CDataMapping` property to direct.
You can set default surface properties on the axes, figure, and root object levels:

```matlab
set(0, 'DefaultSurfaceProperty', PropertyValue)
set(gcf, 'DefaultSurfaceProperty', PropertyValue)
set(gca, 'DefaultSurfaceProperty', PropertyValue)
```

where `Property` is the name of the surface property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.

**See Also**

- ColorSpec, patch, pcolor, surf
- Representing a Matrix as a Surface for examples
- “Surface and Mesh Creation” on page 1-104 and “Object Creation” on page 1-101 for related functions
- Surface Properties for property descriptions
Surface Properties

Purpose

Surface properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The set and get commands enable you to set and query the values of properties.

To change the default values of properties, see Setting Default Property Values.

See “Core Graphics Objects” for general information about this type of object.

Surface Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces {} enclose default values.

AlphaData
m-by-n matrix of double or uint8

\textit{The transparency data.} A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The \texttt{AlphaData} can be of class double or uint8.

MATLAB software determines the transparency in one of three ways:

- Using the elements of \texttt{AlphaData} as transparency values (\texttt{AlphaDataMapping} set to \texttt{none})
- Using the elements of \texttt{AlphaData} as indices into the current alphamap (\texttt{AlphaDataMapping} set to \texttt{direct})
- Scaling the elements of \texttt{AlphaData} to range between the minimum and maximum values of the axes \texttt{ALim} property (\texttt{AlphaDataMapping} set to \texttt{scaled}, the default)
Surface Properties

AlphaDataMapping

none | direct | {scaled}

*Transparency mapping method.* This property determines how MATLAB interprets indexed alpha data. This property can be any of the following:

- **none** — The transparency values of AlphaData are between 0 and 1 or are clamped to this range (the default).
- **scaled** — Transform the AlphaData to span the portion of the alphamap indicated by the axes ALim property, linearly mapping data values to alpha values.
- **direct** — use the AlphaData as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from 1 to length(alphamap). MATLAB maps values less than 1 to the first alpha value in the alphamap, and values greater than length(alphamap) to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If AlphaData is an array of uint8 integers, then the indexing begins at 0 (i.e., MATLAB maps a value of 0 to the first alpha value in the alphamap).

AmbientStrength

scalar >= 0 and <= 1

*Strength of ambient light.* This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes AmbientLightColor property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the surface DiffuseStrength and SpecularStrength properties.
Surface Properties

Annotation

hg.Annotation object Read Only

Control the display of surface objects in legends. The Annotation property enables you to specify whether this surface object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the surface object is displayed in a figure legend:

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Represent this surface object in a legend (default)</td>
</tr>
<tr>
<td>off</td>
<td>Do not include this surface object in a legend</td>
</tr>
<tr>
<td>children</td>
<td>Same as on because surface objects do not have children</td>
</tr>
</tbody>
</table>

Setting the IconDisplayStyle property

These commands set the IconDisplayStyle of a graphics object with handle hobj to off:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property
Surface Properties

See “Controlling Legends” for more information and examples.

**BackFaceLighting**

unlit | lit | reverselit

*Face lighting control.* This property determines how faces are lit when their vertex normals point away from the camera.

- unlit — Face is not lit.
- lit — Face is lit in normal way.
- reverselit — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See “Back Face Lighting” for an example.

**BeingDeleted**

on | {off} Read Only

*This object is being deleted.* The **BeingDeleted** property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the **BeingDeleted** property to on when the object’s delete function callback is called (see the **DeleteFcn** property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted, and therefore, can check the object’s **BeingDeleted** property before acting.

**BusyAction**

cancel | {queue}

*Callback routine interruption.* The **BusyAction** property enables you to control how MATLAB handles events that potentially
interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- **cancel** — Discard the event that attempted to execute a second callback routine.
- **queue** — Queue the event that attempted to execute a second callback routine until the current callback finishes.

**ButtonDownFcn**

function handle, cell array containing function handle and additional arguments, or string (not recommended)

**Button press callback function.** A callback function that executes whenever you press a mouse button while the pointer is over the surface object.

See the figure’s **SelectionType** property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property). For example, the following function takes different action depending on what type of selection was made:

```matlab
function button_down(src,evnt)
    % src - the object that is the source of the event
    % evnt - empty for this property
    sel_typ = get(gcbf,'SelectionType')
    switch sel_typ
        case 'normal'
```
Suppose h is the handle of a surface object and that the button_down function is on your MATLAB path. The following statement assigns the function above to the ButtonDownFcn:

```
set(h, 'ButtonDownFcn', @button_down)
```

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

### CData

**matrix (of type double)**

**Vertex colors.** A matrix containing values that specify the color at every point in ZData.

### Mapping CData to a Colormap

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see caxis) or interpreted directly as indices into the colormap, depending on the setting of the CDataMapping property.

### CData as True Color
Surface Properties

True color defines an RGB value for each vertex. If the coordinate data (XData, for example) are contained in \(m\)-by-\(n\) matrices, then CData must be an \(m\)-by-\(n\)-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

**Texturemapping the Surface FaceColor**

If you set the FaceColor property to texturemap, CData does not need to be the same size as ZData, but must be of type double or uint8. In this case, MATLAB maps CData to conform to the surface defined by ZData.

CDataMapping

\{scaled\} | direct

*Direct or scaled color mapping.* This property determines how MATLAB interprets indexed color data used to color the surface. (If you use true color specification for CData, this property has no effect.)

- scaled — Transform the color data to span the portion of the colormap indicated by the axes CLim property, linearly mapping data values to colors. See the caxis reference page for more information on this mapping.

- direct — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to length(colormap). MATLAB maps values less than 1 to the first color in the colormap, and values greater than length(colormap) to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

Children

matrix of handles

Always the empty matrix; surface objects have no children.
Surface Properties

Clipping
{on} | off

*Clipping to axes rectangle.* When Clipping is on, MATLAB does not display any portion of the surface that is outside the axes rectangle.

CreateFcn
function handle, cell array containing function handle and additional arguments, or string (not recommended)

*Callback function executed during object creation.* This property defines a callback function that executes when MATLAB creates a surface object. You must define this property as a default value for surfaces or set the CreateFcn property during object creation.

For example, the following statement creates a surface (assuming x, y, z, and c are defined), and executes the function referenced by the function handle @myCreateFcn.

```
surface(x,y,z,c,'CreateFcn',@myCreateFcn)
```

MATLAB executes this routine after setting all surface properties. Setting this property on an existing surface object has no effect.

The handle of the object whose CreateFcn is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root CallbackObject property, which you can query using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

DeleteFcn
function handle, cell array containing function handle and additional arguments, or string (not recommended)
Delete surface callback function. A callback function that executes when you delete the surface object (e.g., when you issue a delete command or clear the axes clf or figure clf). For example, the following function displays object property data before the object is deleted.

```matlab
function delete_fcn(src,evnt)
    % src - the object that is the source of the event
    % evnt - empty for this property
    obj_tp = get(src,'Type');
    disp([obj_tp, ' object deleted'])
    disp('Its user data is:')
    disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object’s properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property)

The handle of the object whose DeleteFcn is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root CallbackObject property, which you can query using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

DiffuseStrength

scalar >= 0 and <= 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes.
You can also set the intensity of the ambient and specular components of the light on the surface object. See the AmbientStrength and SpecularStrength properties.

DisplayName

String (default is empty string)

String used by legend for this surface object. The legend function uses the string defined by the DisplayName property to label this surface object in the legend.

- If you specify string arguments with the legend function, DisplayName is set to this surface object’s corresponding string and that string is used for the legend.
- If DisplayName is empty, legend creates a string of the form, ['data' 'n'], where n is the number assigned to the object based on its location in the list of legend entries. However, legend does not set DisplayName to this string.
- If you edit the string directly in an existing legend, DisplayName is set to the edited string.
- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.
- To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See “Controlling Legends” for more examples.

EdgeAlpha

{scalar = 1} | flat | interp

Transparency of the surface edges. This property can be any of the following:

- scalar — A single non-Nan scalar value between 0 and 1 that controls the transparency of all the edges of the object.
1 (the default) means fully opaque and 0 means completely transparent.

- **flat** — The alpha data (AlphaData) value for the first vertex of the face determines the transparency of the edges.

- **interp** — Linear interpolation of the alpha data (AlphaData) values at each vertex determines the transparency of the edge.

Note that you must specify AlphaData as a matrix equal in size to ZData to use flat or interp EdgeAlpha.

**EdgeColor**

{ColorSpec} | none | flat | interp

*Color of the surface edge.* This property determines how MATLAB colors the edges of the individual faces that make up the surface:

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default EdgeColor is black. See ColorSpec for more information on specifying color.

- **none** — Edges are not drawn.

- **flat** — The CData value of the first vertex for a face determines the color of each edge.
Surface Properties

- **interp** — Linear interpolation of the CData values at the face vertices determines the edge color.

**EdgeLighting**

{none} | flat | gouraud | phong

*Algorithm used for lighting calculations.* This property selects the algorithm used to calculate the effect of light objects on surface edges. Choices are

- **none** — Lights do not affect the edges of this object.
- **flat** — The effect of light objects is uniform across each edge of the surface.
- **gouraud** — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- **phong** — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**EraseMode**

{normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase surface objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
Surface Properties

- **none** — Do not erase the surface when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode` *none*, you cannot print it because MATLAB stores no information about its former location.

- **xor** — Draw and erase the surface by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the surface does not damage the color of the objects behind it. However, surface color depends on the color of the screen behind it and is correctly colored only when over the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`.

- **background** — Erase the surface by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased object, but surface objects are always properly colored.

**Printing with Nonnormal Erase Modes**

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing nonnormal mode objects.

**FaceAlpha**

```matlab
{scalar = 1} | flat | interp | texturemap
```

*Transparency of the surface faces.* This property can be any of the following:
- **scalar** — A single non-NaN scalar value between 0 and 1 that controls the transparency of all the faces of the object. 1 (the default) means fully opaque and 0 means completely transparent (invisible).

- **flat** — The values of the alpha data (AlphaData) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.

- **interp** — Bilinear interpolation of the alpha data (AlphaData) at each vertex determines the transparency of each face.

- **texturemap** — Use transparency for the texture map.

Note that you must specify AlphaData as a matrix equal in size to ZData to use flat or interp FaceAlpha.

**FaceColor**

ColorSpec | none | {flat} | interp | texturemap

*Color of the surface face.* This property can be any of the following:

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See ColorSpec for more information on specifying color.

- **none** — Do not draw faces. Note that edges are drawn independently of faces.

- **flat** — The values of CData determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.

- **interp** — Bilinear interpolation of the values at each vertex (the CData) determines the coloring of each face.

- **texturemap** — Texture map the CData to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example.)

**FaceLighting**

{none} | flat | gouraud | phong
Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on the surface. Choices are

- **none** — Lights do not affect the faces of this object.
- **flat** — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
- **gouraud** — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- **phong** — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

HandleVisibility
{on} | callback | off

Control access to object’s handle by command-line users and GUIs. This property determines when an object’s handle is visible in its parent’s list of children. This property is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback routine invokes a function that could potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle’s visibility is restricted using `callback` or `off`, the object’s handle does not appear in its parent’s `Children` property, figures do not appear in the root’s `CurrentFigure` property, objects do not appear in the root’s `CallbackObject` property or in the figure’s `CurrentObject` property, and axes do not appear in their parent’s `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

**HitTest**
  
  `{on} | off

  *Selectable by mouse click.* `HitTest` determines if the surface can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the surface. If `HitTest` is `off`, clicking on the surface selects the object below it (which may be the axes containing it).

**Interruptible**

  `{on} | off
Callback routine interruption mode. The Interruptible property controls whether a surface callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the ButtonDownFcn are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback routine only when it encounters a drawnow, figure, getframe, or pause command in the routine. See the BusyAction property for related information.

LineStyle
{-} | -- | : | -. | none

Edge line type. This property determines the line style used to draw surface edges. The available line styles are shown in this table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solid line (default)</td>
</tr>
<tr>
<td></td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

LineWidth
scalar

Edge line width. The width of the lines in points used to draw surface edges. The default width is 0.5 points (1 point = 1/72 inch).

Marker
marker symbol (see table)

Marker symbol. The Marker property specifies symbols that are displayed at vertices. You can set values for the Marker property independently from the LineStyle property.
You can specify these markers.

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>o</td>
<td>Circle</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

**MarkerEdgeColor**

none | {auto} | flat | ColorSpec

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- none specifies no color, which makes nonfilled markers invisible.
- auto uses the same color as the *EdgeColor* property.
- flat uses the *CData* value of the vertex to determine the color of the maker edge.
Surface Properties

- ColorSpec defines a single color to use for the edge (see ColorSpec for more information).

MarkerFaceColor
{none} | auto | flat | ColorSpec

*Marker face color.* The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- none makes the interior of the marker transparent, allowing the background to show through.
- auto uses the axes Color for the marker face color.
- flat uses the CData value of the vertex to determine the color of the face.
- ColorSpec defines a single color to use for all markers on the surface (see ColorSpec for more information).

MarkerSize
size in points

*Marker size.* A scalar specifying the marker size, in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 the specified marker size.

MeshStyle
{both} | row | column

*Row and column lines.* This property specifies whether to draw all edge lines or just row or column edge lines.

- both draws edges for both rows and columns.
- row draws row edges only.
- column draws column edges only.
NormalMode
{auto} | manual

*MATLAB generated or user-specified normal vectors.* When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to `manual` and does not generate its own data. See also the `VertexNormals` property.

Parent
handle of axes, hggroup, or hgtransform

*Parent of surface object.* This property contains the handle of the surface object’s parent. The parent of a surface object is the axes, hggroup, or hgtransform object that contains it.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

Selected
on | {off}

*Is object selected?* When this property is `on`, MATLAB displays a dashed bounding box around the surface if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

SelectionHighlight
{on} | off

*Objects are highlighted when selected.* When the `Selected` property is `on`, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

SpecularColorReflectance
scalar in the range 0 to 1
**Surface Properties**

*Color of specularly reflected light.* When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the light object Color property). The proportions vary linearly for values in between.

**SpecularExponent**

scalar $\geq 1$

*Harshness of specular reflection.* This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

**SpecularStrength**

scalar $\geq 0$ and $\leq 1$

*Intensity of specular light.* This property sets the intensity of the specular component of the light falling on the surface. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surface object. See the AmbientStrength and DiffuseStrength properties. Also see the material function.

**Tag**

string

*User-specified object label.* The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

**Type**

string (read only)
Class of the graphics object. The class of the graphics object. For surface objects, Type is always the string 'surface'.

UIContextMenu
handle of a uicontextmenu object

Associate a context menu with the surface. Assign this property the handle of a uicontextmenu object created in the same figure as the surface. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click over the surface.

UserData
matrix

User-specified data. Any matrix you want to associate with the surface object. MATLAB does not use this data, but you can access it using the set and get commands.

VertexNormals
vector or matrix

Surface normal vectors. This property contains the vertex normals for the surface. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

Visible
{on} | off

Surface object visibility. By default, all surfaces are visible. When set to off, the surface is not visible, but still exists, and you can query and set its properties.

XData
vector or matrix
Surface Properties

X-coordinates. The x-position of the surface points. If you specify a row vector, surface replicates the row internally until it has the same number of columns as ZData.

YData
vector or matrix

Y-coordinates. The y-position of the surface points. If you specify a row vector, surface replicates the row internally until it has the same number of rows as ZData.

ZData
matrix

Z-coordinates. The z-position of the surfaceplot data points. See the Description section for more information.
Surfaceplot Properties

Purpose

Define surfaceplot properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The set and get commands enable you to set and query the values of properties.

Note that you cannot define default properties for surfaceplot objects.
See Plot Objects for information on surfaceplot objects.

Surfaceplot Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces {} enclose default values.

AlphaData

m-by-n matrix of double or uint8

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The AlphaData can be of class double or uint8.

MATLAB software determines the transparency in one of three ways:

- Using the elements of AlphaData as transparency values (AlphaDataMapping set to none)
- Using the elements of AlphaData as indices into the current alphamap (AlphaDataMapping set to direct)
- Scaling the elements of AlphaData to range between the minimum and maximum values of the axes ALim property (AlphaDataMapping set to scaled, the default)

AlphaDataMapping

{none} | direct | scaled
Surfaceplot Properties

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. It can be any of the following:

- **none** — The transparency values of AlphaData are between 0 and 1 or are clamped to this range (the default).
- **scaled** — Transform the AlphaData to span the portion of the alphamap indicated by the axes ALim property, linearly mapping data values to alpha values.
- **direct** — Use the AlphaData as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from 1 to length(alphamap). MATLAB maps values less than 1 to the first alpha value in the alphamap, and values greater than length(alphamap) to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If AlphaData is an array of uint8 integers, then the indexing begins at 0 (i.e., MATLAB maps a value of 0 to the first alpha value in the alphamap).

**AmbientStrength**

scalar >= 0 and <= 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes AmbientLightColor property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the surfaceplot DiffuseStrength and SpecularStrength properties.

**Annotation**

hg.Annotation object Read Only
Control the display of surfaceplot objects in legends. The Annotation property enables you to specify whether this surfaceplot object is represented in a figure legend.

Querying the Annotation property returns the handle of an hg.Annotation object. The hg.Annotation object has a property called LegendInformation, which contains an hg.LegendEntry object.

Once you have obtained the hg.LegendEntry object, you can set its IconDisplayStyle property to control whether the surfaceplot object is displayed in a figure legend:

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Represent this surfaceplot object in a legend (default)</td>
</tr>
<tr>
<td>off</td>
<td>Do not include this surfaceplot object in a legend</td>
</tr>
<tr>
<td>children</td>
<td>Same as on because surfaceplot objects do not have children</td>
</tr>
</tbody>
</table>

Setting the IconDisplayStyle property

These commands set the IconDisplayStyle of a graphics object with handle hobj to off:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See “Controlling Legends” for more information and examples.
Surfaceplot Properties

BackFaceLighting

unlit | lit | reverselit

*Face lighting control.* This property determines how faces are lit when their vertex normals point away from the camera.

- **unlit** — Face is not lit.
- **lit** — Face is lit in normal way.
- **reverselit** — Face is lit as if the vertex pointed towards the camera.

This property is useful for discriminating between the internal and external surfaces of an object. See Back Face Lighting for an example.

BeingDeleted

on | {off} Read Only

*This object is being deleted.* The **BeingDeleted** property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the **BeingDeleted** property to **on** when the object’s delete function callback is called (see the **DeleteFcn** property). It remains set to **on** while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions might not need to perform actions on objects if the objects are going to be deleted, and therefore, can check the object’s **BeingDeleted** property before acting.

BusyAction

cancel | {queue}

*Callback routine interruption.* The **BusyAction** property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function
executing, callbacks invoked subsequently always attempt to interrupt it.

If the **Interruptible** property of the object whose callback is executing is set to **on** (the default), then interruption occurs at the next point where the event queue is processed. If the **Interruptible** property is **off**, the **BusyAction** property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- **cancel** — Discard the event that attempted to execute a second callback routine.
- **queue** — Queue the event that attempted to execute a second callback routine until the current callback finishes.

**ButtonDownFcn**
cancel | {queue}

*Callback routine interruption.* The **BusyAction** property enables you to control how MATLAB handles events that potentially interrupt executing callbacks. If there is a callback function executing, callbacks invoked subsequently always attempt to interrupt it.

If the **Interruptible** property of the object whose callback is executing is set to **on** (the default), then interruption occurs at the next point where the event queue is processed. If the **Interruptible** property is **off**, the **BusyAction** property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- **cancel** — Discard the event that attempted to execute a second callback routine.
- **queue** — Queue the event that attempted to execute a second callback routine until the current callback finishes.
Surfaceplot Properties

CData
matrix

Vertex colors. A matrix containing values that specify the color at every point in ZData. If you set the FaceColor property to texturemap, CData does not need to be the same size as ZData. In this case, MATLAB maps CData to conform to the surfaceplot defined by ZData.

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see caxis) or interpreted directly as indices into the colormap, depending on the setting of the CDataMapping property. Note that any non-texture data passed as an input argument must be of type double.

True color defines an RGB value for each vertex. If the coordinate data (XData, for example) are contained in m-by-n matrices, then CData must be an m-by-n-by-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

CDataMapping
{scaled} | direct

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the surfaceplot. (If you use true color specification for CData, this property has no effect.)

• scaled — Transform the color data to span the portion of the colormap indicated by the axes CLim property, linearly mapping data values to colors. See the caxis reference page for more information on this mapping.

• direct — Use the color data as indices directly into the colormap. The color data should then be integer values ranging
from 1 to \text{length}(\text{colormap}). MATLAB maps values less than 1 to the first color in the colormap, and values greater than \text{length}(\text{colormap}) to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

\textbf{CDataMode}

\{auto\} | manual

\textit{Use automatic or user-specified color data values.} If you specify \text{CData}, MATLAB sets this property to \text{manual} and uses the \text{CData} values to color the surfaceplot.

If you set \text{CDataMode} to \text{auto} after having specified \text{CData}, MATLAB resets the color data of the surfaceplot to that defined by \text{ZData}, overwriting any previous values for \text{CData}.

\textbf{CDataSource}

string (MATLAB variable)

\textit{Link CData to MATLAB variable.} Set this property to a MATLAB variable that is evaluated in the base workspace to generate the \text{CData}.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change \text{CData}.

You can use the \text{refreshdata} function to force an update of the object's data. \text{refreshdata} also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call \text{refreshdata}.

See the \text{refreshdata} reference page for more information.
Surfaceplot Properties

**Note** If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Children
matrix of handles

Always the empty matrix; surfaceplot objects have no children.

Clipping
{on} | off

*Clipping to axes rectangle.* When Clipping is on, MATLAB does not display any portion of the surfaceplot that is outside the axes rectangle.

CreateFcn
string or function handle

*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates an object. You must specify the callback during the creation of the object. For example,

```matlab
area(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.
The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcb`. You can also set the intensity of the ambient and specular components of the light on the object. See the `AmbientStrength` and `SpecularStrength` properties.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

**DeleteFcn**

string or function handle

*Callback executed during object deletion.* A callback that executes when this object is deleted (e.g., this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcb`. You can also set the intensity of the ambient and specular components of the light on the object. See the `AmbientStrength` and `SpecularStrength` properties.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

**DiffuseStrength**

scalar >= 0 and <= 1

*Intensity of diffuse light.* This property sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the object. See the `AmbientStrength` and `SpecularStrength` properties.
Surfaceplot Properties

DisplayName
string (default is empty string)

*String used by legend for this surfaceplot object.* The `legend` function uses the string defined by the `DisplayName` property to label this surfaceplot object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this surfaceplot object's corresponding string and that string is used for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, ['data' `n`], where `n` is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add programmatically a legend that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more examples.

EdgeAlpha
{scalar = 1} | flat | interp

*Transparency of the patch and surface edges.* This property can be any of the following:

- **scalar** — A single non-Nan scalar value between 0 and 1 that controls the transparency of all the edges of the object. 1 (the default) means fully opaque and 0 means completely transparent.
- **flat** — The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.
• interp — Linear interpolation of the alpha data (AlphaData) values at each vertex determines the transparency of the edge.

Note that you must specify AlphaData as a matrix equal in size to ZData to use flat or interp EdgeAlpha.

EdgeColor
{ColorSpec} | none | flat | interp

*Color of the surfaceplot edge.* This property determines how MATLAB colors the edges of the individual faces that make up the surface:

• ColorSpec — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default EdgeColor is black. See ColorSpec for more information on specifying color.
• none — Edges are not drawn.
• flat — The CData value of the first vertex for a face determines the color of each edge.

• interp — Linear interpolation of the CData values at the face vertices determines the edge color.
Surfaceplot Properties

**EdgeLighting**

{none} | flat | gouraud | phong

*Algorithm used for lighting calculations.* This property selects the algorithm used to calculate the effect of light objects on surfaceplot edges. Choices are

- **none** — Lights do not affect the edges of this object.
- **flat** — The effect of light objects is uniform across each edge of the surface.
- **gouraud** — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- **phong** — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**EraseMode**

{normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase objects and their children. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- **none** — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing
with **EraseMode none**, you cannot print these objects because MATLAB stores no information about their former locations.

- **xor** — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes **Color** property is set to **none**). That is, it isn’t erased correctly if there are objects behind it.

- **background** — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes **Color** property is set to **none**). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

**Printing with Nonnormal Erase Modes**

MATLAB always prints figures as if the **EraseMode** of all objects is **normal**. This means graphics objects created with **EraseMode** set to none, xor, or background can look different on screen than on paper. On screen, MATLAB can mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes **Color** property. Set the figure background color with the figure **Color** property.

You can use the MATLAB **getframe** command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

**FaceAlpha**

{**scalar** = 1} | flat | interp | texturemap
Surfaceplot Properties

**Transparency of the surfaceplot faces.** This property can be any of the following:

- **scalar** — A single non-NaN scalar value between 0 and 1 that controls the transparency of all the faces of the object. 1 (the default) means fully opaque and 0 means completely transparent (invisible).

- **flat** — The values of the alpha data (AlphaData) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.

- **interp** — Bilinear interpolation of the alpha data (AlphaData) at each vertex determines the transparency of each face.

- **texturemap** — Use transparency for the texture map.

Note that you must specify AlphaData as a matrix equal in size to ZData to use flat or interp FaceAlpha.

**FaceColor**

ColorSpec | none | {flat} | interp

**Color of the surfaceplot face.** This property can be any of the following:

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See ColorSpec for more information on specifying color.

- **none** — Do not draw faces. Note that edges are drawn independently of faces.

- **flat** — The values of CData determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.

- **interp** — Bilinear interpolation of the values at each vertex (the CData) determines the coloring of each face.
Surfaceplot Properties

- `texturemap` — Texture map the `Cdata` to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example for `surface`.)

**FaceLighting**

- `{none} | flat | gouraud | phong

*Algorithm used for lighting calculations.* This property selects the algorithm used to calculate the effect of light objects on the surface. Choices are

- `none` — Lights do not affect the faces of this object.

- `flat` — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.

- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.

- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**HandleVisibility**

- `{on} | callback | off

*Control access to object's handle by command-line users and GUIs.* This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible when `HandleVisibility` is on.

- `callback` — Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to
protect GUIs from command-line users, while allowing callback routines to have access to object handles.

- **off** — Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback invokes a function that might potentially damage the GUI (such as evaluating a user-typed string) and so temporarily hides its own handles during the execution of that function.

**Functions Affected by Handle Visibility**

When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `(gca, gcf, gco, newplot, cla, clf, and close`.

**Properties Affected by Handle Visibility**

When a handle’s visibility is restricted using `callback` or `off`, the object’s handle does not appear in its parent’s `Children` property, figures do not appear in the root’s `CurrentFigure` property, objects do not appear in the root’s `CallbackObject` property or in the figure’s `CurrentObject` property, and axes do not appear in their parent’s `CurrentAxes` property.

**Overriding Handle Visibility**

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

**Handle Validity**

Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties and pass it to any function that operates on handles.
**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

**HitTest**

{on} | off

*Selectable by mouse click.* HitTest determines whether this object can become the current object (as returned by the gco command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If HitTest is off, clicking this object selects the object below it (which is usually the axes containing it).

**Interruptible**

{on} | off

*Callback routine interruption mode.* The Interruptible property controls whether an object’s callback can be interrupted by callbacks invoked subsequently.

Only callbacks defined for the ButtonDownFcn property are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the BusyAction property for related information.

Setting Interruptible to on allows any graphics object’s callback to interrupt callback routines originating from a bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**LineStyle**

{-} | – | : | - | none
**Surfaceplot Properties**

*Line style.* This property specifies the line style of the object. Available line styles are shown in the following table.

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>--</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

You can use `LineStyle none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

**LineWidth**
scalar

*The width of linear objects and edges of filled areas.* Specify this value in points (1 point = $\frac{1}{72}$ inch). The default `LineWidth` is 0.5 points.

**Marker**
character (see table)

*Marker symbol.* The `Marker` property specifies the type of markers that are displayed at plot vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include those shown in the following table.

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>0</td>
<td>Circle</td>
</tr>
</tbody>
</table>
## Surfaceplot Properties

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

### MarkerEdgeColor

none | {auto} | flat | ColorSpec

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- **none** specifies no color, which makes nonfilled markers invisible.
- **auto** uses the same color as the EdgeColor property.
- **flat** uses the CData value of the vertex to determine the color of the marker edge.
- **ColorSpec** defines a single color to use for the edge (see ColorSpec for more information).

### MarkerFaceColor

{none} | auto | flat | ColorSpec
**Surfaceplot Properties**

*Marker face color.* The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- **none** makes the interior of the marker transparent, allowing the background to show through.
- **auto** uses the axes **Color** for the marker face color.
- **flat** uses the **CData** value of the vertex to determine the color of the face.
- **ColorSpec** defines a single color to use for all markers on the surfaceplot (see **ColorSpec** for more information).

**MarkerSize**

`size in points`

*Marker size.* A scalar specifying the size of the marker in points. The default value for **MarkerSize** is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

**MeshStyle**

`{both} | row | column`

*Row and column lines.* This property specifies whether to draw all edge lines or just row or column edge lines.

- **both** draws edges for both rows and columns.
- **row** draws row edges only.
- **column** draws column edges only.

**NormalMode**

`{auto} | manual`

*MATLAB generated or user-specified normal vectors.* When this property is **auto**, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals,
MATLAB sets this property to manual and does not generate its own data. See also the VertexNormals property.

**Parent**

handle of parent axes, hggroup, or hgtransform

*Parent of this object.* This property contains the handle of the object’s parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

**Selected**

on | {off}

*Is object selected?* When you set this property to on, MATLAB displays selection "handles" at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

**SelectionHighlight**

{on} | off

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

**SpecularColorReflectance**

scalar in the range 0 to 1

*Color of specularly reflected light.* When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source.
Surfaceplot Properties

When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the light object Color property). The proportions vary linearly for values in between.

SpecularExponent
scalar >= 1

*Harshness of specular reflection.* This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

SpecularStrength
scalar >= 0 and <= 1

*Intensity of specular light.* This property sets the intensity of the specular component of the light falling on the surface. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surfaceplot object. See the AmbientStrength and DiffuseStrength properties. Also see the material function.

Tag
string

*User-specified object label.* The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define Tag as any string.

For example, you might create an areaseries object and set the Tag property.

```
t = area(Y, 'Tag', 'area1')
```
When you want to access objects of a given type, you can use `findobj` to find the object’s handle. The following statement changes the FaceColor property of the object whose Tag is `area1`.

```matlab
set(findobj('Tag','area1'),'FaceColor','red')
```

### Type

**string (read only)**

*Class of the graphics object.* The class of the graphics object. For surfaceplot objects, `Type` is always the string `'surface'`.

### UIContextMenu

**handle of an uicontextmenu object**

*Associate a context menu with this object.* Assign this property the handle of an `uicontextmenu` object created in the object’s parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object.

### UserData

**array**

*User-specified data.* This property can be any data you want to associate with this object (including cell arrays and structures). The object does not set values for this property, but you can access it using the `set` and `get` functions.

### VertexNormals

**vector or matrix**

*Surfaceplot normal vectors.* This property contains the vertex normals for the surfaceplot. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.
Surfaceplot Properties

Visible
{on} | off

Visibility of this object and its children. By default, a new object’s visibility is on. This means all children of the object are visible unless the child object’s Visible property is set to off. Setting an object’s Visible property to off prevents the object from being displayed. However, the object still exists and you can set and query its properties.

XData
vector or matrix

X-coordinates. The x-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of columns as ZData.

XDataMode
{auto} | manual

Use automatic or user-specified x-axis values. If you specify XData (by setting the XData property or specifying the x input argument), MATLAB sets this property to manual and uses the specified values to label the x-axis.

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks to 1:size(YData,1) or to the column indices of the ZData, overwriting any previous values for XData.

XDataSource
string (MATLAB variable)

Link XData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.
MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change XData.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

**YData**

vector or matrix

*Y-coordinates.* The y-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of rows as ZData.

**YDataMode**

{`auto` | `manual`}

*Use automatic or user-specified x-axis values.* If you specify XData, MATLAB sets this property to `manual`.

If you set `YDataMode` to `auto` after having specified `YData`, MATLAB resets the y-axis ticks and y-tick labels to the row indices of the ZData, overwriting any previous values for `YData`.

**YDataSource**

string (MATLAB variable)
Surfaceplot Properties

Link YData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change YData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

ZData
matrix

Z-coordinates. The z-position of the surfaceplot data points. See the Description section for more information.

ZDataSource
string (MATLAB variable)

Link ZData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the ZData.
MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change **ZData**.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Purpose
Surface plot with colormap-based lighting

GUI Alternatives
To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax
surfl(Z)
surfl(...,'light')
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)

Description
The surfl function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

surfl(Z) and surfl(X,Y,Z) create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. X, Y, and Z are vectors or matrices that define the x, y, and z components of a surface.

surfl(...,'light') produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, surfl(...,'cdata'), which changes the color data for the surface to be the reflectance of the surface.

surfl(...,s) specifies the direction of the light source. s is a two- or three-element vector that specifies the direction from a surface to a light source. s = [sx sy sz] or s = [azimuth elevation]. The default s is 45° counterclockwise from the current view direction.

surfl(X,Y,Z,s,k) specifies the reflectance constant. k is a four-element vector defining the relative contributions of ambient light,
diffuse reflection, specular reflection, and the specular shine coefficient.
k = [ka kd ks shine] and defaults to [.55, .6, .4, 10].

h = surfl(...) returns a handle to a surface graphics object.

Remarks

surfl does not accept complex inputs.
For smoother color transitions, use colormaps that have linear intensity variations (e.g., gray, copper, bone, pink).
The ordering of points in the X, Y, and Z matrices defines the inside and outside of parametric surfaces. If you want the opposite side of the surface to reflect the light source, use surfl(X', Y', Z'). Because of the way surface normal vectors are computed, surfl requires matrices that are at least 3-by-3.

Examples

View peaks using colormap-based lighting.

```matlab
[x, y] = meshgrid(-3:1/8:3);
z = peaks(x, y);
surfl(x, y, z);
shading interp
colormap(gray);
axis([-3 3 -3 3 -8 8])
```
To plot a lighted surface from a view direction other than the default,

```matlab
view([10 10])
grid on
hold on
surfl(peaks)
shading interp
colormap copper
hold off
```
See Also  
colormap, shading, light

“Surface and Mesh Creation” on page 1-104 for functions related to surfaces

“Lighting” on page 1-108 for functions related to lighting
**Purpose**
Compute and display 3-D surface normals

**Syntax**

```
surfnorm(Z)
surfnorm(X,Y,Z)
[Nx,Ny,Nz] = surfnorm(…)
```

**Description**
The `surfnorm` function computes surface normals for the surface defined by `X`, `Y`, and `Z`. The surface normals are unnormalized and valid at each vertex. Normals are not shown for surface elements that face away from the viewer.

`surfnorm(Z)` and `surfnorm(X,Y,Z)` plot a surface and its surface normals. `Z` is a matrix that defines the `z` component of the surface. `X` and `Y` are vectors or matrices that define the `x` and `y` components of the surface.

`[Nx,Ny,Nz] = surfnorm(…)` returns the components of the three-dimensional surface normals for the surface.

**Remarks**
`surfnorm` does not accept complex inputs.

The direction of the normals is reversed by calling `surfnorm` with transposed arguments:

```
surfnorm(X',Y',Z')
```

`surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.
Algorithm

The surface normals are based on a bicubic fit of the data in X, Y, and Z. For each vertex, diagonal vectors are computed and crossed to form the normal.

Examples

Plot the normal vectors for a truncated cone.

```
x, y, z = cylinder(1:10);
surfnorm(x, y, z)
axis([-12 12 -12 12 -0.1 1])
```

See Also

surf, quiver3

for related functions
svd

Purpose
Singular value decomposition

Syntax
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
[U,S,V] = svd(X, 'econ')

Description
The svd command computes the matrix singular value decomposition.
s = svd(X) returns a vector of singular values.
[U,S,V] = svd(X) produces a diagonal matrix S of the same dimension
as X, with nonnegative diagonal elements in decreasing order, and
unitary matrices U and V so that X = U*S*V'.

[U,S,V] = svd(X,0) produces the “economy size” decomposition. If X
is m-by-n with m > n, then svd computes only the first n columns of U
and S is n-by-n.

[U,S,V] = svd(X, 'econ') also produces the “economy size”
decomposition. If X is m-by-n with m >= n, it is equivalent to svd(X,0).
For m < n, only the first m columns of V are computed and S is m-by-m.

Examples
For the matrix

X =
| 1  2 |
| 3  4 |
| 5  6 |
| 7  8 |

the statement

[U,S,V] = svd(X)

produces

U =
| -0.1525  -0.8226  -0.3945  -0.3800 |

2-3624
The economy size decomposition generated by

\[ [U, S, V] = \text{svd}(X, 0) \]

produces

\[
U =
\begin{bmatrix}
-0.1525 & -0.8226 \\
-0.3499 & -0.4214 \\
-0.5474 & -0.0201 \\
-0.7448 & 0.3812
\end{bmatrix}
\]

\[
S =
\begin{bmatrix}
14.2691 & 0 \\
0 & 0.6268 \\
0 & 0 \\
0 & 0
\end{bmatrix}
\]

\[
V =
\begin{bmatrix}
-0.6414 & 0.7672 \\
-0.7672 & -0.6414
\end{bmatrix}
\]

Algorithm

svd uses the LAPACK routines listed in the following table to compute the singular value decomposition.
svd

<table>
<thead>
<tr>
<th></th>
<th>Real</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>X double</td>
<td>DGESVD</td>
<td>ZGESVD</td>
</tr>
<tr>
<td>X single</td>
<td>SGESVD</td>
<td>CGESVD</td>
</tr>
</tbody>
</table>

**Diagnostics**

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

Solution will not converge.

**References**

**Purpose**
Find singular values and vectors

**Syntax**

s = svds(A)
s = svds(A,k)
s = svds(A,k,sigma)
s = svds(A,k,'L')
s = svds(A,k,sigma,options)
[U,S,V] = svds(A,...)
[U,S,V,flag] = svds(A,...)

**Description**

\( s = \text{svds}(A) \) computes the six largest singular values and associated singular vectors of matrix \( A \). If \( A \) is \( m \)-by-\( n \), \( \text{svds}(A) \) manipulates eigenvalues and vectors returned by \( \text{eigs}(B) \), where \( B = [\text{sparse}(m,m) A; A' \text{sparse}(n,n)] \), to find a few singular values and vectors of \( A \). The positive eigenvalues of the symmetric matrix \( B \) are the same as the singular values of \( A \).

\( s = \text{svds}(A,k) \) computes the \( k \) largest singular values and associated singular vectors of matrix \( A \).

\( s = \text{svds}(A,k,sigma) \) computes the \( k \) singular values closest to the scalar shift \( sigma \). For example, \( s = \text{svds}(A,k,0) \) computes the \( k \) smallest singular values and associated singular vectors.

\( s = \text{svds}(A,k,'L') \) computes the \( k \) largest singular values (the default).

\( s = \text{svds}(A,k,sigma,options) \) sets some parameters (see \( \text{eigs} \)):

**Option Structure Fields and Descriptions**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Parameter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.tol</td>
<td>Convergence tolerance: ( \text{norm}(AV-US,1) \leq tol \times \text{norm}(A,1) )</td>
<td>1e-10</td>
</tr>
<tr>
<td>options.maxit</td>
<td>Maximum number of iterations</td>
<td>300</td>
</tr>
<tr>
<td>options.disp</td>
<td>Number of values displayed each iteration</td>
<td>0</td>
</tr>
</tbody>
</table>
[U,S,V] = svds(A,...) returns three output arguments, and if A is m-by-n:

- U is m-by-k with orthonormal columns
- S is k-by-k diagonal
- V is n-by-k with orthonormal columns
- U*S*V' is the closest rank k approximation to A

[U,S,V,flag] = svds(A,...) returns a convergence flag. If eigs converged then norm(A*V-U*S,1) <= tol*norm(A,1) and flag is 0. If eigs did not converge, then flag is 1.

Note svds is best used to find a few singular values of a large, sparse matrix. To find all the singular values of such a matrix, svd(full(A)) will usually perform better than svds(A,min(size(A))).

Algorithm

svds(A,k) uses eigs to find the k largest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0].

svds(A,k,0) uses eigs to find the 2k smallest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0], and then selects the k positive eigenvalues and their eigenvectors.

Example

west0479 is a real 479-by-479 sparse matrix. svd calculates all 479 singular values. svds picks out the largest and smallest singular values.

load west0479
s = svd(full(west0479))
s1 = svds(west0479,4)
ss = svds(west0479,6,0)

These plots show some of the singular values of west0479 as computed by svd and svds.
The largest singular value of `west0479` can be computed a few different ways:

\[
\text{svds}(\text{west0479}, 1) = \quad 3.189517598808622e+05 \\
\max(\text{svd}(\text{full}(\text{west0479}))) = \quad 3.18951759880862e+05 \\
\|\text{full}(\text{west0479})\| = \quad 3.189517598808623e+05 \\
\text{normest}(\text{west0479}) = \quad 3.189385666549991e+05
\]

and estimated:

\[
\text{normest}(\text{west0479}) = \quad 3.189385666549991e+05
\]

See Also

`svd`, `eigs`
**Purpose**
Swap byte ordering

**Syntax**

\[ Y = \text{swapbytes}(X) \]

**Description**

\( Y = \text{swapbytes}(X) \) reverses the byte ordering of each element in array \( X \), converting little-endian values to big-endian (and vice versa). The input array must contain all full, noncomplex, numeric elements.

**Examples**

**Example 1**

Reverse the byte order for a scalar 32-bit value, changing hexadecimal 12345678 to 78563412:

```matlab
A = \text{uint32}(\text{hex2dec('12345678'))};

B = \text{dec2hex}(\text{swapbytes}(A))
B =
78563412
```

**Example 2**

Reverse the byte order for each element of a 1-by-4 matrix:

```matlab
X = \text{uint16}([0 1 128 65535])
X =
0 1 128 65535

Y = \text{swapbytes}(X);
Y =
0 256 32768 65535
```

Examining the output in hexadecimal notation shows the byte swapping:

```matlab
\text{format hex}

X, Y
X =
0000 0001 0080 ffff
```
Y =
   0000  0100  8000  ffff

**Example 3**

Create a three-dimensional array `A` of 16-bit integers and then swap the bytes of each element:

```matlab
format hex
A = uint16(magic(3) * 150);
A(:,:,2) = A * 40;

A
A(:,:,1) =
   04b0  0096  0384
   01c2  02ee  041a
   0258  0546  012c
A(:,:,2) =
   bb80  1770  8ca0
   4650  7530  a410
   5dc0  d2f0  2ee0

swapbytes(A)
an(:,:,1) =
   b004  9600  8403
   c201  ee02  1a04
   5802  4605  2c01
an(:,:,2) =
   80bb  7017  a08c
   5046  3075  10a4
   c05d  f0d2  e02e
```

**See Also**

typecast
**Purpose**

Switch among several cases, based on expression

**Syntax**

```plaintext
switch switch_expr
    case case_expr
        statement, ..., statement
    case {case_expr1, case_expr2, case_expr3, ...}
        statement, ..., statement
    otherwise
        statement, ..., statement
end
```

**Discussion**

The `switch` statement syntax is a means of conditionally executing code. In particular, `switch` executes one set of statements selected from an arbitrary number of alternatives. Each alternative is called a case, and consists of

- The case statement
- One or more case expressions
- One or more statements

In its basic syntax, `switch` executes the statements associated with the first case where `switch_expr == case_expr`. When the case expression is a cell array (as in the second case above), the `case_expr` matches if any of the elements of the cell array matches the switch expression. If no case expression matches the switch expression, then control passes to the otherwise case (if it exists). After the case is executed, program execution resumes with the statement after the `end`.

The `switch_expr` can be a scalar or a string. A scalar `switch_expr` matches a `case_expr` if `switch_expr == case_expr`. A string `switch_expr` matches a `case_expr` if `strcmp(switch_expr, case_expr)` returns logical 1 (true).
**Note for C Programmers** Unlike the C language `switch` construct, the MATLAB `switch` does not “fall through.” That is, `switch` executes only the first matching case; subsequent matching cases do not execute. Therefore, `break` statements are not used.

**Examples** To execute a certain block of code based on what the string, `method`, is set to,

```matlab
method = 'Bilinear';

switch lower(method)
    case {'linear','bilinear'}
        disp('Method is linear')
    case 'cubic'
        disp('Method is cubic')
    case 'nearest'
        disp('Method is nearest')
    otherwise
        disp('Unknown method.')
end

Method is linear
```

**See Also** `case`, `otherwise`, `end`, `if`, `else`, `elseif`, `while`
**Purpose**
Symmetric approximate minimum degree permutation

**Syntax**

```matlab
p = symamd(S)
p = symamd(S,knobs)
[p,stats] = symamd(...)
```

**Description**

`p = symamd(S)` for a symmetric positive definite matrix `S`, returns the permutation vector `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. To find the ordering for `S`, `symamd` constructs a matrix `M` such that `spones(M'*M) = spones(S)`, and then computes `p = colamd(M)`. The `symamd` function may also work well for symmetric indefinite matrices.

`S` must be square; only the strictly lower triangular part is referenced.

`p = symamd(S,knobs)` where `knobs` is a scalar. If `S` is `n`-by-`n`, rows and columns with more than `knobs*n` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs = spparms('wh_frac')`.

`[p,stats] = symamd(...)` produces the optional vector `stats` that provides data about the ordering and the validity of the matrix `S`.

- `stats(1)` Number of dense or empty rows ignored by `symamd`
- `stats(2)` Number of dense or empty columns ignored by `symamd`
- `stats(3)` Number of garbage collections performed on the internal data structure used by `symamd` (roughly of size `8.4*nnz(tril(S,-1)) + 9n` integers)
- `stats(4)` 0 if the matrix is valid, or 1 if invalid
- `stats(5)` Rightmost column index that is unsorted or contains duplicate entries, or 0 if no such column exists
- `stats(6)` Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or 0 if no such row index exists
- `stats(7)` Number of duplicate and out-of-order row indices
Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to symamd. For this reason, symamd verifies that S is valid:

- If a row index appears two or more times in the same column, symamd ignores the duplicate entries, continues processing, and provides information about the duplicate entries in stats(4:7).

- If row indices in a column are out of order, symamd sorts each column of its internal copy of the matrix S (but does not repair the input matrix S), continues processing, and provides information about the out-of-order entries in stats(4:7).

- If S is invalid in any other way, symamd cannot continue. It prints an error message, and returns no output arguments (p or stats).

The ordering is followed by a symmetric elimination tree post-ordering.

**Examples**

Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the symrcm reference page.

```matlab
B = bucky+4*speye(60);
r = symrcm(B);
p = symamd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R,4), title('B(r,r)')
subplot(2,2,2), spy(S,4), title('B(s,s)')
subplot(2,2,3), spy(chol(R),4), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S),4), title('chol(B(s,s))')
```
Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

**See Also**
colamd, colperm, spparms, symrcm, amd

**References**
The authors of the code for symamd are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert,
Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory.
Sparse Matrix Algorithms Research at the University of Florida:
http://www.cise.ufl.edu/research/sparse/
**Purpose**
Symbolic factorization analysis

**Syntax**

```matlab
count = symbfact(A)
count = symbfact(A,'sym')
count = symbfact(A,'col')
count = symbfact(A,'row')
count = symbfact(A,'lo')
[count,h,parent,post,R] = symbfact(...)
[count,h,parent,post,L] = symbfact(A,type,'lower')
```

**Description**

- `count = symbfact(A)` returns the vector of row counts of \( R = \text{chol}(A'\cdot A) \). `symbfact` should be much faster than `chol(A)`.
- `count = symbfact(A,'sym')` is the same as `count = symbfact(A)`.
- `count = symbfact(A,'col')` returns row counts of \( R = \text{chol}(A'\cdot A) \) (without forming it explicitly).
- `count = symbfact(A,'row')` returns row counts of \( R = \text{chol}(A^2) \).
- `count = symbfact(A,'lo')` is the same as `count = symbfact(A)` and uses `tril(A)`.

- `[count,h,parent,post,R] = symbfact(...)` has several optional return values.

The flop count for a subsequent Cholesky factorization is `sum(count.^2)`

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>Height of the elimination tree</td>
</tr>
<tr>
<td>parent</td>
<td>The elimination tree itself</td>
</tr>
<tr>
<td>post</td>
<td>Postordering of the elimination tree</td>
</tr>
<tr>
<td>R</td>
<td>0-1 matrix having the structure of <code>chol(A)</code> for the symmetric case, <code>chol(A'\cdot A)</code> for the 'col' case, or <code>chol(A\cdot A')</code> for the 'row' case.</td>
</tr>
</tbody>
</table>
symbfact(A) and symbfact(A,'sym') use the upper triangular part of A (triu(A)) and assume the lower triangular part is the transpose of the upper triangular part. symbfact(A,'lo') uses tril(A) instead.

[count,h,parent,post,L] = symbfact(A,type,'lower') where type is one of 'sym','col','row', or 'lo' returns a lower triangular symbolic factor L=R'. This form is quicker and requires less memory.

See Also
chol, etree, treelayout
**Purpose**
Symmetric LQ method

**Syntax**

```matlab
x = symmlq(A,b)
symmlq(A,b,tol)
symmlq(A,b,tol,maxit)
symmlq(A,b,tol,maxit,M)
symmlq(A,b,tol,maxit,M1,M2)
symmlq(A,b,tol,maxit,M1,M2,x0)
[x,flag] = symmlq(A,b,...)
[x,flag,relres] = symmlq(A,b,...)
[x,flag,relres,iter] = symmlq(A,b,...)
[x,flag,relres,iter,resvec] = symmlq(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)
```

**Description**

`x = symmlq(A,b)` attempts to solve the system of linear equations

\[ A*x = b \]

for \( x \). The \( n \)-by-\( n \) coefficient matrix \( A \) must be symmetric but need not be positive definite. It should also be large and sparse. The column vector \( b \) must have length \( n \). \( A \) can be a function handle \texttt{afun} such that \( \texttt{afun(x)} \) returns \( A*x \). See “Function Handles” in the MATLAB Programming documentation for more information.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function \texttt{afun}, as well as the preconditioner function \texttt{mfun} described below, if necessary.

If \texttt{symmlq} converges, a message to that effect is displayed. If \texttt{symmlq} fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual \( \text{norm}(b-A*x)/\text{norm}(b) \) and the iteration number at which the method stopped or failed.

\texttt{symmlq(A,b,tol)} specifies the tolerance of the method. If \texttt{tol} is [], then \texttt{symmlq} uses the default, 1e-6.

\texttt{symmlq(A,b,tol,maxit)} specifies the maximum number of iterations. If \texttt{maxit} is [], then \texttt{symmlq} uses the default, \( \min(n,20) \).
symmlq(A, b, tol, maxit, M) and symmlq(A, b, tol, maxit, M1, M2) use the symmetric positive definite preconditioner M or M = M1*M2 and effectively solve the system \( \text{inv}(\sqrt{M}) A \text{inv}(\sqrt{M}) y = \text{inv}(\sqrt{M}) b \) for y and then return \( x = \text{inv}(\sqrt{M}) y \). If M is [], then symmlq applies no preconditioner. M can be a function handle mfun such that mfun(x) returns \( M \x \).

symmlq(A, b, tol, maxit, M1, M2, x0) specifies the initial guess. If x0 is [], then symmlq uses the default, an all-zero vector.

\[ [x, \text{flag}] = \text{symmlq}(A, b, \ldots) \] also returns a convergence flag.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>symmlq converged to the desired tolerance tol within maxit iterations.</td>
</tr>
<tr>
<td>1</td>
<td>symmlq iterated maxit times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>Preconditioner M was ill-conditioned.</td>
</tr>
<tr>
<td>3</td>
<td>symmlq stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during symmlq became too small or too large to continue computing.</td>
</tr>
<tr>
<td>5</td>
<td>Preconditioner M was not symmetric positive definite.</td>
</tr>
</tbody>
</table>

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.

\[ [x, \text{flag}, \text{relres}] = \text{symmlq}(A, b, \ldots) \] also returns the relative residual \( \text{norm}(b - A*x)/\text{norm}(b) \). If flag is 0, relres \( \leq \) tol.

\[ [x, \text{flag}, \text{relres}, \text{iter}] = \text{symmlq}(A, b, \ldots) \] also returns the iteration number at which x was computed, where 0 \( \leq \) iter \( \leq \) maxit.

\[ [x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{symmlq}(A, b, \ldots) \] also returns a vector of estimates of the symmlq residual norms at each iteration, including \( \text{norm}(b - A*x0) \).
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...) also returns a vector of estimates of the conjugate gradients residual norms at each iteration.

Examples

Example 1

n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50; M1 = spdiags(4*on,0,n,n);

x = symmlq(A,b,tol,maxit,M1);
symmlq converged at iteration 49 to a solution with relative residual 4.3e-015

Example 2

This example replaces the matrix A in Example 1 with a handle to a matrix-vector product function afun. The example is contained in an M-file run_symmlq that

- Calls symmlq with the function handle @afun as its first argument.
- Contains afun as a nested function, so that all variables in run_symmlq are available to afun.

The following shows the code for run_symmlq:

```matlab
function x1 = run_symmlq
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
```
x1 = symmlq(@afun,b,tol,maxit,M1);

function y = afun(x)
    y = 4 * x;
    y(2:n) = y(2:n) - 2 * x(1:n-1);
    y(1:n-1) = y(1:n-1) - 2 * x(2:n);
end
end

When you enter
x1=run_symmlq;

MATLAB software displays the message

symmlq converged at iteration 49 to a solution with relative residual 4.3e-015

**Example 3**

Use a symmetric indefinite matrix that fails with pcg.

A = diag([20:-1:1,-1:-1:-20]);
b = sum(A,2); % The true solution is the vector of all ones.
x = pcg(A,b); % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired tolerance 1e-006 because a scalar quantity became too small or too large to continue computing.
The iterate returned (number 0) has relative residual 1

However, symmlq can handle the indefinite matrix A.

x = symmlq(A,b,1e-6,40);
symmlq converged at iteration 39 to a solution with relative residual 1.3e-007

**See Also**

bicg, bicgstab, cgs, lsqr, gmres, minres, pcg, qmr

function_handle (@), mldivide (\)
References


**Purpose**  
Sparse reverse Cuthill-McKee ordering

**Syntax**  
r = symrcm(S)

**Description**  
r = symrcm(S) returns the symmetric reverse Cuthill-McKee ordering of S. This is a permutation r such that S(r,r) tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric S.

For a real, symmetric sparse matrix, S, the eigenvalues of S(r,r) are the same as those of S, but eig(S(r,r)) probably takes less time to compute than eig(S).

**Algorithm**  
The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

**Examples**  
The statement

```matlab
B = bucky;
```

uses an M-file in the demos toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name bucky), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together. With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows

```matlab
subplot(1,2,1), spy(B), title('B')
```

The reverse Cuthill-McKee ordering is obtained with
\begin{verbatim}
p = symrcm(B);
R = B(p,p);

The spy plot shows a much narrower bandwidth.

subplot(1,2,2), spy(R), title('B(p,p)')
\end{verbatim}

This example is continued in the reference pages for \texttt{symamd}.

The bandwidth can also be computed with

\begin{verbatim}
[i,j] = find(B);
bw = max(i-j) + 1;
\end{verbatim}

The bandwidths of \texttt{B} and \texttt{R} are 35 and 12, respectively.

**See Also**

\texttt{colmnd, colperm}, \texttt{symamd}

**References**


**Purpose**
Determine symbolic variables in expression

**Syntax**
```
symvar 'expr'
s = symvar('expr')
```

**Description**
symvar 'expr' searches the expression, expr, for identifiers other than i, j, pi, inf, nan, eps, and common functions. symvar displays those variables that it finds or, if no such variable exists, displays an empty cell array, {}.

```
s = symvar('expr')
```
returns the variables in a cell array of strings, s. If no such variable exists, s is an empty cell array.

**Examples**
symvar finds variables beta1 and x, but skips pi and the cos function.

```
symvar 'cos(pi*x - beta1)'
```

```
ans =

'betal'
'x'
```

**See Also**
findstr
**Purpose**
Synchronize and resample two timeseries objects using common time vector

**Syntax**

\[
[ts1 \ ts2] = \text{synchronize}(ts1, ts2, 'SynchronizeMethod')
\]

**Description**

\[
[ts1 \ ts2] = \text{synchronize}(ts1, ts2, 'SynchronizeMethod')
\]
creates two new timeseries objects by synchronizing \(ts1\) and \(ts2\) using a common time vector. The string 'SynchronizeMethod' defines the method for synchronizing the timeseries and can be one of the following:

- **'Union'** — Resample timeseries objects using a time vector that is a union of the time vectors of \(ts1\) and \(ts2\) on the time range where the two time vectors overlap.
- **'Intersection'** — Resample timeseries objects on a time vector that is the intersection of the time vectors of \(ts1\) and \(ts2\).
- **'Uniform'** — Requires an additional argument as follows:

\[
[ts1 \ ts2] = \text{synchronize}(ts1, ts2, 'Uniform', 'Interval', value)
\]

This method resamples time series on a uniform time vector, where value specifies the time interval between the two samples. The uniform time vector is the overlap of the time vectors of \(ts1\) and \(ts2\). The interval units are assumed to be the smaller units of \(ts1\) and \(ts2\).

You can specify additional arguments by using property-value pairs:

- **'InterpMethod'**: Forces the specified interpolation method (over the default method) for this synchronize operation. Can be either a string, 'linear' or 'zoh', or a \text{tsdata.interpolation} object that contains a user-defined interpolation method.
- **'QualityCode'**: Integer (between -128 and 127) used as the quality code for both time series after the synchronization.
- 'KeepOriginalTimes': Logical value (true or false) indicating whether the new time series should keep the original time values. For example,

```matlab
ts1 = timeseries([1 2],[datestr(now); datestr(now+1)]);
ts2 = timeseries([1 2],[datestr(now-1); datestr(now)]);
```

Note that ts1.timeinfo.StartDate is one day after ts2.timeinfo.StartDate. If you use

```matlab
[ts1 ts2] = synchronize(ts1,ts2,'union');
```

the ts1.timeinfo.StartDate is changed to match ts2.TimeInfo.StartDate and ts1.Time changes to 1.

But if you use

```matlab
[ts1 ts2] = synchronize(ts1,ts2,'union','KeepOriginalTimes',true);
```

ts1.timeinfo.StartDate is unchanged and ts1.Time is still 0.

- 'tolerance': Real number used as the tolerance for differentiating two time values when comparing the ts1 and ts2 time vectors. The default tolerance is 1e-10. For example, when the sixth time value in ts1 is 5+(1e-12) and the sixth time value in ts2 is 5-(1e-13), both values are treated as 5 by default. To differentiate those two times, you can set 'tolerance' to a smaller value such as 1e-15, for example.

**See Also**
timeseries
Purpose

Two ways to call MATLAB functions

Description

You can call MATLAB functions using either *command syntax* or *function syntax*, as described below.

**Command Syntax**

A function call in this syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname arg1 arg2 ... argn
```

Command syntax does not allow you to obtain any values that might be returned by the function. Attempting to assign output from the function to a variable using command syntax generates an error. Use function syntax instead.

Examples of command syntax:

```
save mydata.mat x y z
import java.awt.Button java.lang.String
```

Arguments are treated as string literals. See the examples below, under “Argument Passing” on page 2-3651.

**Function Syntax**

A function call in this syntax consists of the function name followed by one or more arguments separated by commas and enclosed in parentheses:

```
functionname(arg1, arg2, ..., argn)
```

You can assign the output of the function to one or more output values. When assigning to more than one output variable, separate the variables by commas or spaces and enclose them in square brackets ([]):

```
[out1,out2,...,outn] = functionname(arg1, arg2, ..., argn)
```

Examples of function syntax:
copyfile('srcfile', '..\mytests', 'writable')
[x1,x2,x3,x4] = deal(A{:})

Arguments are passed to the function by value. See the examples below, under “Argument Passing” on page 2-3651.

**Argument Passing**

When calling a function using command syntax, MATLAB passes the arguments as string literals. When using function syntax, arguments are passed by value.

In the following example, assign a value to A and then call disp on the variable to display the value passed. Calling disp with command syntax passes the variable name, 'A':

```
A = pi;
disp A
A
```

while function syntax passes the value assigned to A:

```
A = pi;
disp(A)
3.1416
```

The next example passes two strings to strcmp for comparison. Calling the function with command syntax compares the variable names, 'str1' and 'str2':

```
str1 = 'one';   str2 = 'one';
strcmp str1 str2
ans =
0        (unequal)
```

while function syntax compares the values assigned to the variables, 'one' and 'one':

```
str1 = 'one';   str2 = 'one';
strcmp(str1, str2)
```
ans =

    1       (equal)

**Passing Strings**

When using the function `syntax` to pass a string literal to a function, you must enclose the string in single quotes, ('string'). For example, to create a new directory called `myapptests`, use

```plaintext
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```plaintext
dirname = 'myapptests';
mkdir(dirname)
```

**See Also**

`mlint`
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Execute operating system command and return result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>system('command')</td>
</tr>
<tr>
<td></td>
<td>[status, result] = system('command')</td>
</tr>
</tbody>
</table>
**Description**

`system('command')` calls upon the operating system to run `command`, for example `dir` or `ls` or a UNIX shell script, and directs the output to the MATLAB software. If `command` runs successfully, `ans` is 0. If `command` fails or does not exist on your operating system, `ans` is a nonzero value and an explanatory message appears.

```
[status, result] = system('command')
```
calls upon the operating system to run `command`, and directs the output to MATLAB. If `command` runs successfully, `status` is 0 and `result` contains the output from `command`. If `command` fails or does not exist on your operating system, `status` is a nonzero value and `result` contains an explanatory message.

**Note** Running `system` on a Microsoft Windows platform with a command that relies on the current directory fails when the current directory is specified using a UNC pathname because DOS does not support UNC pathnames. When this happens, MATLAB returns the error:

```
??? Error using ==> system DOS commands may not be executed when the current directory is a UNC pathname.
```

To work around this limitation, change the directory to a mapped drive prior to running `system` or a function that calls `system`.

**Examples**

On a Windows system, display the current directory by accessing the operating system.

```
[status currdir] = system('cd')
status =
    0
currdir =
    D:\work\matlab\test
```

**See Also**

! (bang), computer, dos, perl, unix, winopen

26. UNIX is a registered trademark of The Open Group in the United States and other countries.
“Running External Programs” in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**
Tangent of argument in radians

**Syntax**
\[ Y = \tan(X) \]

**Description**
The \( \tan \) function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.
\[ Y = \tan(X) \] returns the circular tangent of each element of \( X \).

**Examples**
Graph the tangent function over the domain \(-\pi/2 < x < \pi/2\).

```matlab
x = (-pi/2)+0.01:0.01:(pi/2)-0.01;
p = plot(x,tan(x)), grid on
```

The expression \( \tan(\pi/2) \) does not evaluate as infinite but as the reciprocal of the floating point accuracy \( \text{eps} \) since \( \pi \) is only a floating-point approximation to the exact value of \( \pi \).

**Definition**
The tangent can be defined as
\[ \tan(z) = \frac{\sin(z)}{\cos(z)} \]

**Algorithm**  
\( \tan \) uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see [http://www.netlib.org](http://www.netlib.org).

**See Also**  
tand, tanh, atan, atan2, atand, atanh
Purpose  
Tangent of argument in degrees

Syntax  
Y = tand(X)

Description  
Y = tand(X) is the tangent of the elements of X, expressed in degrees. For odd integers n, tand(n*90) is infinite, whereas tan(n*pi/2) is large but finite, reflecting the accuracy of the floating point value of pi.

See Also  
tan, tanh, atan, atan2, atand, atanh
**Purpose**
Hyperbolic tangent

**Syntax**

\[ Y = \tanh(X) \]

**Description**
The \( \tanh \) function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

\[ Y = \tanh(X) \] returns the hyperbolic tangent of each element of \( X \).

**Examples**
Graph the hyperbolic tangent function over the domain \(-5 \leq x \leq 5\).

```matlab
x = -5:0.01:5;
plot(x,tanh(x)), grid on
```

**Definition**
The hyperbolic tangent can be defined as

\[ \tanh(z) = \frac{\sinh(z)}{\cosh(z)} \]
**tanh**

**Algorithm**

tanh uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

**See Also**

atan, atan2, tan
**Purpose**
Compress files into tar file

**Syntax**

tar(tarfilename,files)
tar(tarfilename,files,rootdir)
entrynames = tar(...)

**Description**
tar(tarfilename,files) creates a tar file named tarfilename from the list of files and directories specified in files. Relative paths are stored in the tar file, but absolute paths are not. Directories recursively include all of their content.

tarfilename is a string specifying the name of the tar file. MATLAB appends the .tar extension if tarfilename has no extension. The tarfilename extension can end in .tgz or .gz. In this case, tarfilename is gzipped.

files is a string or cell array of strings containing the list of files or directories included in tarfilename. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current directory or with an absolute path.
Directories must be specified relative to the current directory or with absolute paths. On UNIX\textsuperscript{27} systems, directories can also start with \texttt{-/} or \texttt{-username/}, which expands to the current user’s home directory or the specified user’s home directory, respectively. The wildcard character * can be used when specifying files or directories, except when relying on the MATLAB path to resolve a file name or partial path name.

\texttt{tar(tarfilename,files,rootdir)} allows the path for \texttt{files} to be specified relative to \texttt{rootdir} rather than the current directory.

\texttt{entrynames = tar(...)} returns a string cell array of the names of the files contained in \texttt{tarfilename}. If \texttt{files} contains a relative path, \texttt{entrynames} also contains the relative path.

**Example**

Tar all files in the current directory to the file backup.tgz.

\begin{verbatim}
    tar('backup.tgz','.');
\end{verbatim}

**See Also**
gzip, gunzip, untar, unzip, zip

\textsuperscript{27} UNIX is a registered trademark of The Open Group in the United States and other countries.
**Purpose**  
Name of system’s temporary directory

**Syntax**  
tmp_dir = tempdir

**Description**  
tmp_dir = tempdir returns the name of the system’s temporary directory, if one exists. This function does not create a new directory. See “Opening Temporary Files and Directories” for more information.

**See Also**  
tempname
**tempname**

**Purpose**
Unique name for temporary file

**Syntax**
tmp_nam = tempname

**Description**
tmp_nam = tempname returns a unique string, tmp_nam, suitable for use as a temporary filename.

**Note**
The filename that tempname generates is not guaranteed to be unique; however, it is likely to be so.

See “Opening Temporary Files and Directories” for more information.

**See Also**
tempdir
**Purpose**
Tetrahedron mesh plot

**Syntax**
- `tetramesh(T,X,c)`
- `tetramesh(T,X)`
- `h = tetramesh(...)`
- `tetramesh(...,'param','value','param','value'...)`

**Description**
`tetramesh(T,X,c)` displays the tetrahedrons defined in the m-by-4 matrix T as mesh. T is usually the output of `delaunayn`. A row of T contains indices into X of the vertices of a tetrahedron. X is an n-by-3 matrix, representing n points in 3 dimension. The tetrahedron colors are defined by the vector C, which is used as indices into the current colormap.

**Note**
If T is the output of `delaunay3`, then X is the concatenation of the `delaunay3` input arguments x, y, z interpreted as column vectors, i.e.,

\[
X = [x(:) \ y(:) \ z(:)].
\]

tetramesh(T,X) uses \(C = 1:m\) as the color for the m tetrahedrons. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

\[h = \text{tetramesh}(...)\]
returns a vector of tetrahedron handles. Each element of h is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch 'Visible' property 'on' or 'off'.

\[\text{tetramesh}(..., 'param', 'value', 'param', 'value' ...)\]
allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to 0.9. You can overwrite this value by using the property name/property value pair ('FaceAlpha', value) where value is a number between 0 and 1. See Patch Properties for information about the available properties.
Examples

Generate a 3-dimensional Delaunay tessellation, then use `tetramesh` to visualize the tetrahedrons that form the corresponding simplex.

d = [-1 1];
[x,y,z] = meshgrid(d,d,d); % A cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
X = [x(:) y(:) z(:)];
Tes = delaunayn(X)

Tes =
    9  1  5  6
   3  9  1  5
   2  9  1  6
   2  3  9  4
   2  3  9  1
   7  9  5  6
   7  3  9  5
   8  7  9  6
   8  2  9  6
   8  2  9  4
   8  3  9  4
   8  7  3  9
	etramesh(Tes,X);camorbit(20,0)
See Also
delaunayn, patch, Patch Properties, trimesh, trisurf
Purpose

Produce TeX format from character string

Syntax

texlabel(f)
texlabel(f,'literal')

Description

texlabel(f) converts the MATLAB expression f into the TeX equivalent for use in text strings. It processes Greek variable names (e.g., lambda, delta, etc.) into a string that is displayed as actual Greek letters.

texlabel(f,'literal') prints Greek variable names as literals.

If the string is too long to fit into a figure window, then the center of the expression is replaced with a tilde ellipsis (~~~).

Examples

You can use texlabel as an argument to the title, xlabel, ylabel, zlabel, and text commands. For example,

    title(texlabel('sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2)'))

By default, texlabel translates Greek variable names to the equivalent Greek letter. You can select literal interpretation by including the literal argument. For example, compare these two commands.

    text(.5,.5,...
       texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)'))
    text(.25,.25,...
       texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)','literal'))
See Also

text, title, xlabel, ylabel, zlabel, the text String property

“Annotating Plots” on page 1-94 for related functions
**Purpose**
Create text object in current axes

**Syntax**
- `text(x,y,'string')`
- `text(x,y,z,'string')`
- `text(x,y,z,'string',PropertyName,PropertyValue....)`
- `text('PropertyName',PropertyValue....)`
- `h = text(...)`

**Description**
`text` is the low-level function for creating text graphics objects. Use `text` to place character strings at specified locations.

- `text(x,y,'string')` adds the string in quotes to the location specified by the point `(x,y)`. `x` and `y` must be numbers of class `double`.
- `text(x,y,z,'string')` adds the string in 3-D coordinates. `x`, `y`, and `z` must be numbers of class `double`.
- `text(x,y,z,'string',PropertyName,PropertyValue....)` adds the string in quotes to the location defined by the coordinates and uses the values for the specified text properties. See the text property list at the end of this page for a list of text properties.
- `text('PropertyName',PropertyValue....)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

- `h = text(...)` returns a column vector of handles to text objects, one handle per object. All forms of the `text` function optionally return this output argument.

See the String property for a list of symbols, including Greek letters.

**Remarks**

**Position Text Within the Axes**
The default text units are the units used to plot data in the graph. Specify the text location coordinates (the `x`, `y`, and `z` arguments) in the data units of the current graph (see “Examples” on page 2-3672. You can use other units to position the text by setting the text Units property to normalized or one of the nonrelative units (pixels, inches, centimeters, points).
Note that the Axes Units property controls the positioning of the Axes within the figure and is not related to the axes data units used for graphing.

The Extent, VerticalAlignment, and HorizontalAlignment properties control the positioning of the character string with regard to the text location point.

If the coordinates are vectors, text writes the string at all locations defined by the list of points. If the character string is an array the same length as $x$, $y$, and $z$, text writes the corresponding row of the string array at each point specified.

**Multiline Text**

When specifying strings for multiple text objects, the string can be

- A cell array of strings
- A padded string matrix

Each element of the specified string array creates a different text object.

When specifying the string for a single text object, cell arrays of strings and padded string matrices result in a text object with a multiline string, while vertical slash characters are not interpreted as separators and result in a single line string containing vertical slashes.

**Behavior of the Text Function**

text is a low-level function that accepts property name/property value pairs as input arguments. However, the convenience form,

```
    text(x,y,z,'string')
```

is equivalent to

```
    text('Position',[x,y,z],'String','string')
```

You can specify other properties only as property name/property value pairs. See the text property list at the end of this page for a description
of each property. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the set and get reference pages for examples of how to specify these data types).

text does not respect the setting of the figure or axes NextPlot property. This allows you to add text objects to an existing axes without setting hold to on.

**Examples**

The statements

```matlab
plot(0:pi/20:2*pi,sin(0:pi/20:2*pi))
text(pi,0,' \leftarrow \sin(\pi)','FontSize',18)
```

annotate the point at (pi,0) with the string \( \sin(\pi) \)
The statement

\[ e^{i \omega \tau} = \cos(\omega \tau) + i \sin(\omega \tau) \]

uses embedded TeX sequences to produce

\[ e^{i \omega \tau} = \cos(\omega \tau) + i \sin(\omega \tau) \]

**Setting Default Properties**

You can set default text properties on the axes, figure, and root object levels:

```matlab
set(0,'DefaultTextProperty',PropertyValue...)  
set(gcf,'DefaultTextProperty',PropertyValue...)```

2-3673
set(gca,'DefaulttextProperty',PropertyValue...) 

Where Property is the name of the text property and PropertyValue is the value you are specifying. Use set and get to access text properties.

See Also

annotation, gtext, int2str, num2str, strings, title, xlabel, ylabel, zlabel

“Object Creation” on page 1-101 for related functions

Text Properties for property descriptions
**Purpose**

Text properties

**Modifying Properties**

You can set and query graphics object properties using the property editor or the set and get commands.

- The Property Editor is an interactive tool that enables you to see and change object property values.

- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see Setting Default Property Values.

See Core Objects for general information about this type of object.

**Text Property Descriptions**

This section lists property names along with the types of values each accepts. Curly braces `{}` enclose default values.

**Annotation**

`hg.Annotation` object Read Only

*Control the display of text objects in legends.* The `Annotation` property enables you to specify whether this text object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains a `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the text object is displayed in a figure legend:
### Text Properties

<table>
<thead>
<tr>
<th>IconDisplayStyle Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Represent this text object in a legend (default)</td>
</tr>
<tr>
<td>off</td>
<td>Do not include this text object in a legend</td>
</tr>
<tr>
<td>children</td>
<td>Same as on because text objects do not have children</td>
</tr>
</tbody>
</table>

**Setting the IconDisplayStyle property**

These commands set the IconDisplayStyle of a graphics object with handle hobj to off:

```matlab
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

**Using the IconDisplayStyle property**

See “Controlling Legends” for more information and examples.

**BackgroundColor**

ColorSpec | {none}

*Color of text extent rectangle.* This property enables you to define a color for the rectangle that encloses the text Extent plus the text Margin. For example, the following code creates a text object that labels a plot and sets the background color to light green.

```matlab
text(3*pi/4,sin(3*pi/4),...  
['sin(3*pi/4) = ',num2str(sin(3*pi/4))],...  
'HorizontalAlignment','center',...)  
'BackgroundColor',[.7 .9 .7]);
```
For additional features, see the following properties:

- **EdgeColor** — Color of the rectangle’s edge (none by default).
- **LineStyle** — Style of the rectangle’s edge line (first set EdgeColor)
- **LineWidth** — Width of the rectangle’s edge line (first set EdgeColor)
- **Margin** — Increase the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the EdgeColor rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the EdgeColor property and the area defined by the BackgroundColor change.

See also Drawing Text in a Box in the MATLAB Graphics documentation for an example using background color with contour labels.
**Text Properties**

**BeingDeleted**

on | {off} read only

*This object is being deleted.* The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB software sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property) It remains set to on while the delete function executes, after which the object no longer exists.

For example, an object’s delete function might call other functions that act on a number of different objects. These functions may not need to perform actions on objects that are going to be deleted, and therefore can check the object’s BeingDeleted property before acting.

**BusyAction**

cancel | {queue}

*Callback routine interruption.* The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, callback routines invoked subsequently always attempt to interrupt it. If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the Interruptible property is set to off, the BusyAction property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback routine.
- `queue` — Queue the event that attempted to execute a second callback routine until the current callback finishes.
ButtonDownFcn

function handle, cell array containing function handle and additional arguments, or string (not recommended)

*Button press callback function.* A callback function that executes whenever you press a mouse button while the pointer is over the text object.

See the figure’s `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property). For example, the following function takes different action depending on what type of selection was made:

```matlab
function button_down(src,evnt)
    % src - the object that is the source of the event
    % evnt - empty for this property
    sel_typ = get(gcaf,'SelectionType')
    switch sel_typ
        case 'normal'
            disp('User clicked left-mouse button')
            set(src,'Selected','on')
        case 'extend'
            disp('User did a shift-click')
            set(src,'Selected','on')
        case 'alt'
            disp('User did a control-click')
            set(src,'Selected','on')
            set(src,'SelectionHighlight','off')
        end
    end
end
```
Suppose \( h \) is the handle of a text object and that the \texttt{button\_down} function is on your MATLAB path. The following statement assigns the function above to the \texttt{ButtonDownFcn}:

\[
\text{set}(h, 'ButtonDownFcn', @button\_down)
\]

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

\textbf{Children}

\texttt{matrix (read only)}

The empty matrix; text objects have no children.

\textbf{Clipping}

\texttt{on | \{off\}}

\textit{Clipping mode}. When \texttt{Clipping} is on, MATLAB does not display any portion of the text that is outside the axes.

\textbf{Color}

\texttt{ColorSpec}

\textit{Text color}. A three-element RGB vector or one of the predefined names, specifying the text color. The default value is black. See \texttt{ColorSpec} for more information on specifying color.

\textbf{CreateFcn}

\texttt{function handle, cell array containing function handle and additional arguments, or string (not recommended)}

\textit{Callback function executed during object creation}. A callback function that executes when MATLAB creates a text object. You must define this property as a default value for text or in a call to the \texttt{text} function that creates a new text object. For example, the statement

\[
\text{set}(0, 'DefaultTextCreateFcn', @text\_create)
\]
defines a default value on the root level that sets the figure
Pointer property to crosshairs whenever you create a text object.
The callback function must be on your MATLAB path when you
execute the above statement.

```matlab
function text_create(src,evnt)
    % src - the object that is the source of the event
    % evnt - empty for this property
    set(gcaf, 'Pointer', 'crosshair')
end
```

MATLAB executes this function after setting all text properties.
Setting this property on an existing text object has no effect. The
function must define at least two input arguments (handle of
object created and an event structure, which is empty for this
property).

The handle of the object whose CreateFcn is being executed is
passed by MATLAB as the first argument to the callback function
and is also accessible through the root CallbackObject property,
which you can query using gcbo.

See “Function Handle Callbacks” for information on how to use
function handles to define the callback function.

DeleteFcn
function handle, cell array containing function handle and
additional arguments, or string (not recommended)

Delete text callback function. A callback function that executes
when you delete the text object (e.g., when you issue a delete
command or clear the axes cla or figure clf). For example, the
following function displays object property data before the object
is deleted.

```matlab
function deleteFcn(src,evnt)
    % src - the object that is the source of the event
    % evnt - empty for this property
```
MATLAB executes the function before deleting the object’s properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property).

The handle of the object whose DeleteFcn is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root CallbackObject property, which you can query using gcbo.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

DisplayName

string (default is empty string)

*String used by legend for this text object.* The `legend` function uses the string defined by the DisplayName property to label this text object in the legend.

- If you specify string arguments with the `legend` function, `DisplayName` is set to this text object’s corresponding string and that string is used for the legend.

- If `DisplayName` is empty, `legend` creates a string of the form, ['data' n], where n is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.

- If you edit the string directly in an existing legend, `DisplayName` is set to the edited string.
If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.

To add programmatically a legend that uses the DisplayName string, call legend with the toggle or show option.

See “Controlling Legends” for more examples.

**EdgeColor**

ColorSpec | {none}

*Color of edge drawn around text extent rectangle plus margin.* This property enables you to specify the color of a box drawn around the text Extent plus the text Margin. For example, the following code draws a red rectangle around text that labels a plot.

```matlab
text(3*pi/4,sin(3*pi/4),...
    '\leftarrow\sin(t) = .707',... 
    'EdgeColor','red');
```

For additional features, see the following properties:
Text Properties

- **BackgroundColor** — Color of the rectangle’s interior (none by default)
- **LineStyle** — Style of the rectangle’s edge line (first set EdgeColor)
- **LineWidth** — Width of the rectangle’s edge line (first set EdgeColor)
- **Margin** — Increases the size of the rectangle by adding a margin to the area defined by the text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the EdgeColor rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the EdgeColor property and the area defined by the BackgroundColor change.

**Editing**

- **on** | **{off}**

*Enable or disable editing mode.* When this property is set to the default off, you cannot edit the text string interactively (i.e., you must change the String property to change the text). When this property is set to on, MATLAB places an insert cursor at the end of the text string and enables editing. To apply the new text string,

1. Press the **Esc** key.
2. Click in any figure window (including the current figure).
3. Reset the Editing property to off.

MATLAB then updates the String property to contain the new text and resets the Editing property to off. You must reset the Editing property to on to resume editing.

**EraseMode**

- **{normal}** | **none** | **xor** | **background**
Erase mode. This property controls the technique MATLAB uses to draw and erase text objects. Alternative erase modes are useful for creating animated sequences where controlling the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- **normal** — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- **none** — Do not erase the text when it is moved or destroyed. While the object is still visible on the screen after erasing with **EraseMode none**, you cannot print it because MATLAB stores no information about its former location.

- **xor** — Draw and erase the text by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. When the text is erased, it does not damage the objects beneath it. However, when text is drawn in **xor** mode, its color depends on the color of the screen beneath it. It is correctly colored only when it is over axes background Color, or the figure background Color if the axes **Color** is set to **none**.

- **background** — Erase the text by drawing it in the axes background Color, or the figure background Color if the axes **Color** is set to **none**. This damages objects that are behind the erased text, but text is always properly colored.

**Printing with Nonnormal Erase Modes**

MATLAB always prints figures as if the **EraseMode** of all objects is set to **normal**. This means graphics objects created with **EraseMode** set to **none**, **xor**, or **background** can look differently on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., performing an XOR of a pixel color with that of the pixel behind it) and ignore three-dimensional
sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing nonnormal mode objects.

**Extent**

position rectangle (read only)

*Position and size of text.* A four-element read-only vector that defines the size and position of the text string

```
[left,bottom,width,height]
```

If the `Units` property is set to `data` (the default), `left` and `bottom` are the x- and y-coordinates of the lower left corner of the text Extent.

For all other values of `Units`, `left` and `bottom` are the distance from the lower left corner of the axes position rectangle to the lower left corner of the text Extent. `width` and `height` are the dimensions of the Extent rectangle. All measurements are in units specified by the `Units` property.

**FontAngle**

{normal} | italic | oblique

*Character slant.* MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

**FontName**

A name, such as `Courier`, or the string `FixedWidth`

*Font family.* A string specifying the name of the font to use for the text object. To display and print properly, this must be a font that your system supports. The default font is Helvetica.
Specifying a Fixed-Width Font

If you want text to use a fixed-width font that looks good in any locale, you should set FontName to the string FixedWidth:

```matlab
set(text_handle,'FontName','FixedWidth')
```

This eliminates the need to hard-code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan where multibyte character sets are used). A properly written MATLAB application that needs to use a fixed-width font should set FontName to FixedWidth (note that this string is case sensitive) and rely on FixedWidthFontName to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root FixedWidthFontName property to the appropriate value for that locale from `startup.m`.

Note that setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

FontSize

size in FontUnits

*Font size.* A value specifying the font size to use for text in units determined by the FontUnits property. The default point size is 10 (1 point = 1/72 inch).

FontWeight

light | {normal} | demi | bold

*Weight of text characters.* MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to bold or demi causes MATLAB to use a bold font.

FontUnits

{points} | normalized | inches | centimeters | pixels
Font size units. MATLAB uses this property to determine the units used by the FontSize property. Normalized units interpret FontSize as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen FontSize accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

Note that if you are setting both the FontSize and the FontUnits in one function call, you must set the FontUnits property first so that MATLAB can correctly interpret the specified FontSize.

HandleVisibility
{on} | callback | off

Control access to object’s handle by command-line users and GUIs. This property determines when an object’s handle is visible in its parent’s list of children. HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is set to on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.
When a handle is not visible in its parent’s list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes \texttt{get}, \texttt{findobj}, \texttt{gca}, \texttt{gcf}, \texttt{gco}, \texttt{newplot}, \texttt{cla}, \texttt{clf}, and \texttt{close}.

When a handle’s visibility is restricted using \texttt{callback} or \texttt{off},

- The object’s handle does not appear in its parent’s \texttt{Children} property.
- Figures do not appear in the root’s \texttt{CurrentFigure} property.
- Objects do not appear in the root’s \texttt{CallbackObject} property or in the figure’s \texttt{CurrentObject} property.
- Axes do not appear in their parent’s \texttt{CurrentAxes} property.

You can set the root \texttt{ShowHiddenHandles} property to \texttt{on} to make all handles visible regardless of their \texttt{HandleVisibility} settings (this does not affect the values of the \texttt{HandleVisibility} properties).

Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

\texttt{HitTest} \hfill \{\texttt{on}} \mid \texttt{off}\hfill

\textit{Selectable by mouse click}. \texttt{HitTest} determines if the text can become the current object (as returned by the \texttt{gco} command and the figure \texttt{CurrentObject} property) as a result of a mouse click on the text. If \texttt{HitTest} is set to \texttt{off}, clicking the text selects the object below it (which is usually the axes containing it).

For example, suppose you define the button down function of an image (see the \texttt{ButtonDownFcn} property) to display text at the location you click with the mouse.
First define the callback routine.

```matlab
function bd_function
    pt = get(gca,'CurrentPoint');
    text(pt(1,1),pt(1,2),pt(1,3),...
        '{\fontsize{20}\oplus} The spot to label',...
        'HitTest','off')
```

Now display an image, setting its ButtonDownFcn property to the callback routine.

```matlab
load earth
image(X,'ButtonDownFcn','bd_function'); colormap(map)
```

When you click the image, MATLAB displays the text string at that location. With HitTest set to off, existing text cannot intercept any subsequent button down events that occur over the text. This enables the image's button down function to execute.

**HorizontalAlignment**

`{left} | center | right`

*Horizontal alignment of text.* This property specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the Position property. The following picture illustrates the alignment options.

HorizontalAlignment viewed with the VerticalAlignment set to middle (the default).

```
Left  Center  Right
```

See the Extent property for related information.
Interpreter
latex | {tex} | none

*Interpret \(\TeX\) instructions.* This property controls whether MATLAB interprets certain characters in the `String` property as \(\TeX\) instructions (default) or displays all characters literally. The options are:

- \latex{} — Supports the full \LaTeX{} markup language.
- \text{} — Supports a subset of plain \TeX{} markup language. See the `String` property for a list of supported \TeX{} instructions.
- none — Displays literal characters.

**Latex Interpreter**

To enable the \LaTeX{} interpreter for text objects, set the `Interpreter` property to `latex`. For example, the following statement displays an equation in a figure at the point \([0.5, 0.5]\), and enlarges the font to 16 points.

```matlab
text('Interpreter','latex',...
    'String','$$\int_0^x\int_y dF(u,v)$$',...
    'Position', [.5 .5],...
    'FontSize', 16)
```

2-3691
Information About Using TEX

The following references may be useful to people who are not familiar with TEX.

- The TEX Users Group home page: http://www.tug.org

**Interruptible**

{on} | off

*Callback routine interruption mode.* The Interruptible property controls whether a text callback routine can be interrupted by subsequently invoked callback routines. Text objects have three properties that define callback routines: ButtonDownFcn,
CreateFcn, and DeleteFcn. See the BusyAction property for information on how MATLAB executes callback routines.

**LineStyle**

{ - | -- | : | -. | none }

*Edge line type.* This property determines the line style used to draw the edges of the text Extent. The available line styles are shown in the following table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>–</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>-.</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

For example, the following code draws a red rectangle with a dotted line style around text that labels a plot.

```matlab
text(3*pi/4,sin(3*pi/4),...    
     '\leftarrow\sin(t) = .707',...    
     'EdgeColor','red',...    
     'LineWidth',2,...    
     'LineStyle',':');
```
For additional features, see the following properties:

- **BackgroundColor** — Color of the rectangle’s interior (none by default)
- **EdgeColor** — Color of the rectangle’s edge (none by default)
- **LineWidth** — Width of the rectangle’s edge line (first set EdgeColor)
- **Margin** — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the EdgeColor rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the EdgeColor property and the area defined by the BackgroundColor change.

**LineWidth**

scalar (points)

*Width of line used to draw text extent rectangle.* When you set the text EdgeColor property to a color (the default is none), MATLAB...
displays a rectangle around the text Extent. Use the LineWidth property to specify the width of the rectangle edge. For example, the following code draws a red rectangle around text that labels a plot and specifies a line width of 3 points:

```matlab
   text(3*pi/4,sin(3*pi/4),...
   '\leftarrow\sin(t) = .707',...
   'EdgeColor','red',...
   'LineWidth',3);
```

For additional features, see the following properties:

- **BackgroundColor** — Color of the rectangle’s interior (none by default)
- **EdgeColor** — Color of the rectangle’s edge (none by default)
- **LineStyle** — Style of the rectangle’s edge line (first set EdgeColor)
- **Margin** — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background
area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

**Margin**

scalar (pixels)

*Distance between the text extent and the rectangle edge.* When you specify a color for the `BackgroundColor` or `EdgeColor` text properties, MATLAB draws a rectangle around the area defined by the text Extent plus the value specified by the Margin. For example, the following code displays a light green rectangle with a 10-pixel margin.

```matlab
text(5*pi/4,sin(5*pi/4),...
    ['sin(5*pi/4) = ',num2str(sin(5*pi/4))],...
    'HorizontalAlignment','center',...
    'BackgroundColor',[.7 .9 .7],...
    'Margin',10);
```

![Image](image.png)
For additional features, see the following properties:

- **BackgroundColor** — Color of the rectangle’s interior (none by default)
- **EdgeColor** — Color of the rectangle’s edge (none by default)
- **LineStyle** — Style of the rectangle’s edge line (first set **EdgeColor**)
- **LineWidth** — Width of the rectangle’s edge line (first set **EdgeColor**)

**See how margin affects text extent properties**

This example enables you to change the values of the **Margin** property and observe the effects on the **BackgroundColor** area and the **EdgeColor** rectangle.

Click to view in editor — This link opens the MATLAB editor with the following example.

Click to run example — Use your scroll wheel to vary the **Margin**.

**Parent**

handle of axes, hggroup, or hgtransform

*Parent of text object.* This property contains the handle of the text object’s parent. The parent of a text object is the axes, hggroup, or hgtransform object that contains it.

See Objects That Can Contain Other Objects for more information on parenting graphics objects.

**Position**

\[x, y, [z]\]

*Location of text.* A two- or three-element vector, \[x, y, [z]\], that specifies the location of the text in three dimensions. If you
omit the z value, it defaults to 0. All measurements are in units specified by the Units property. Initial value is [0 0 0].

**Rotation**

scalar (default = 0)

*Text orientation.* This property determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation).

**Selected**

on | {off}

*Is object selected?* When this property is set to on, MATLAB displays selection handles if the SelectionHighlight property is also set to on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

**SelectionHighlight**

{on} | off

*Objects are highlighted when selected.* When the Selected property is set to on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is set to off, MATLAB does not draw the handles.

**String**

string

*The text string.* Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.
Note The words default, factory, and remove are reserved words that will not appear in a figure when quoted as a normal string. In order to display any of these words individually, type ‘\reserved_word’ instead of ‘reserved_word’.

When the text Interpreter property is set to Tex (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

<table>
<thead>
<tr>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
<th>Character Sequence</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\alpha</td>
<td>α</td>
<td>\upsilon</td>
<td>υ</td>
<td>\sim</td>
<td>~</td>
</tr>
<tr>
<td>\beta</td>
<td>β</td>
<td>\phi</td>
<td>Φ</td>
<td>\leq</td>
<td>≤</td>
</tr>
<tr>
<td>\gamma</td>
<td>γ</td>
<td>\chi</td>
<td>χ</td>
<td>\infty</td>
<td>∞</td>
</tr>
<tr>
<td>\delta</td>
<td>δ</td>
<td>\psi</td>
<td>ψ</td>
<td>\clubsuit</td>
<td>♠</td>
</tr>
<tr>
<td>\epsilon</td>
<td>ε</td>
<td>\omega</td>
<td>Ω</td>
<td>\diamondsuit</td>
<td>♦</td>
</tr>
<tr>
<td>\zeta</td>
<td>ζ</td>
<td>\Gamma</td>
<td>Γ</td>
<td>\heartsuit</td>
<td>♥</td>
</tr>
<tr>
<td>\eta</td>
<td>η</td>
<td>\Delta</td>
<td>Δ</td>
<td>\spadesuit</td>
<td>♠</td>
</tr>
<tr>
<td>\theta</td>
<td>Θ</td>
<td>\Theta</td>
<td>Θ</td>
<td>\lefttrightarrow</td>
<td>↔</td>
</tr>
<tr>
<td>\vartheta</td>
<td>θ</td>
<td>\Lambda</td>
<td>Λ</td>
<td>\leftarrow</td>
<td>←</td>
</tr>
<tr>
<td>\iota</td>
<td>ι</td>
<td>\Xi</td>
<td>Ξ</td>
<td>\uparrow</td>
<td>↑</td>
</tr>
<tr>
<td>\kappa</td>
<td>κ</td>
<td>\Pi</td>
<td>Π</td>
<td>\rightarrow</td>
<td>→</td>
</tr>
<tr>
<td>\lambda</td>
<td>λ</td>
<td>\Sigma</td>
<td>Σ</td>
<td>\downarrow</td>
<td>↓</td>
</tr>
<tr>
<td>\mu</td>
<td>μ</td>
<td>\Upsilon</td>
<td>Υ</td>
<td>\circ</td>
<td>°</td>
</tr>
</tbody>
</table>
You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However,
you can use \fontname in combination with one of the other modifiers:

- \bf — Bold font
- \it — Italic font
- \sl — Oblique font (rarely available)
- \rm — Normal font
- \fontname{\fontname} — Specify the name of the font family to use.
- \fontsize{\fontsize} — Specify the font size in FontUnits.
- \color(colorSpec) — Specify color for succeeding characters

Stream modifiers remain in effect until the end of the string or only within the context defined by braces {}.

**Specifying Text Color in TeX Strings**

Use the \color modifier to change the color of characters following it from the previous color (which is black by default). Syntax is:

- \color{colorname} for the eight basic named colors (red, green, yellow, magenta, blue, black, white), and plus the four Simulink colors (gray, darkGreen, orange, and lightBlue)

  Note that short names (one-letter abbreviations) for colors are not supported by the \color modifier.

- \color[rgb]{r g b} to specify an RGB triplet with values between 0 and 1 as a cell array

For example,

```latex
\text{.1,.5,['}\\text{\fontsize{16}{black \ \color{magenta}{magenta \ ... \\
\color[rgb]{0 \ .5 \ .5}{teal \ \color{red}{red} \ black \ again'}\]})
```

2-3701
### Specifying Subscript and Superscript Characters

The subscript character “\_” and the superscript character “\^” modify the character or substring defined in braces immediately following.

To print the special characters used to define the TeX strings when **Interpreter** is **Tex**, prefix them with the backslash “\” character: \_, \{, \}, \_, \^.

See the “Examples” on page 2-3672 in the text reference page for more information.

When **Interpreter** is set to **none**, no characters in the **String** are interpreted, and all are displayed when the text is drawn.

When **Interpreter** is set to **latex**, MATLAB provides a complete LaTeX interpreter for text objects. See the **Interpreter** property for more information.
Tag

string

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when you are constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

Type

string (read only)

Class of graphics object. For text objects, Type is always the string 'text'.

Units

pixels | normalized | inches | centimeters | points | {data}

Units of measurement. This property specifies the units MATLAB uses to interpret the Extent and Position properties. All units are measured from the lower left corner of the axes plot box.

- Normalized units map the lower left corner of the rectangle defined by the axes to (0,0) and the upper right corner to (1,0,1,0).

- pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

- data refers to the data units of the parent axes as determined by the data graphed (not the axes Units property, which controls the positioning of the axes within the figure window).

If you change the value of Units, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume Units is set to the default value.
UserData
matrix

*User-specified data.* Any data you want to associate with the text object. MATLAB does not use this data, but you can access it using set and get.

UIContextMenu
handle of a uicontextmenu object

*Associate a context menu with the text.* Assign this property the handle of a uicontextmenu object created in the same figure as the text. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click over the text.

VerticalAlignment
top | cap | {middle} | baseline | bottom

*Vertical alignment of text.* This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the Position property. The possible values mean

- **top** — Place the top of the string’s *Extent* rectangle at the specified *y*-position.
- **cap** — Place the string so that the top of a capital letter is at the specified *y*-position.
- **middle** — Place the middle of the string at the specified *y*-position.
- **baseline** — Place font baseline at the specified *y*-position.
- **bottom** — Place the bottom of the string’s *Extent* rectangle at the specified *y*-position.

The following picture illustrates the alignment options.
Text VerticalAlignment property viewed with the HorizontalAlignment property set to left (the default).

Visible
{on} | off

Text visibility. By default, all text is visible. When set to off, the text is not visible, but still exists, and you can query and set its properties.
textread

**Purpose**
Read data from text file; write to multiple outputs

**Note**
textread is not recommended. Use textscan to read data from a text file.

**Graphical Interface**
As an alternative to textread, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

**Syntax**

```
[A,B,C,...] = textread('filename', 'format')
[A,B,C,...] = textread('filename','format',N)
[...] = textread(...,'param','value',...)
```

**Description**

[A,B,C,...] = textread('filename', 'format') reads data from the file 'filename' into the variables A, B, C, and so on, using the specified format, until the entire file is read. The filename and format inputs are strings, each enclosed in single quotes. textread is useful for reading text files with a known format. textread handles both fixed and free format files.

**Note**
When reading large text files, reading from a specific point in a file, or reading file data into a cell array rather than multiple outputs, you might prefer to use the textscan function.

textread matches and converts groups of characters from the input. Each input field is defined as a string of non-white-space characters that extends to the next white-space or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated white-space characters are treated as one.

The format string determines the number and types of return arguments. The number of return arguments is the number of items in the format string. The format string supports a subset of the conversion specifiers and conventions of the C language fscanf routine.
Values for the format string are listed in the table below. White-space characters in the format string are ignored.

<table>
<thead>
<tr>
<th>format</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literals (ordinary characters)</td>
<td>Ignore the matching characters. For example, in a file that has Dept followed by a number (for department number), to skip the Dept and read only the number, use 'Dept' in the format string.</td>
<td>None</td>
</tr>
<tr>
<td>%d</td>
<td>Read a signed integer value.</td>
<td>Double array</td>
</tr>
<tr>
<td>%u</td>
<td>Read an integer value.</td>
<td>Double array</td>
</tr>
<tr>
<td>%f</td>
<td>Read a floating-point value.</td>
<td>Double array</td>
</tr>
<tr>
<td>%s</td>
<td>Read a white-space or delimiter-separated string.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>%q</td>
<td>Read a double quoted string, ignoring the quotes.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>%c</td>
<td>Read characters, including white space.</td>
<td>Character array</td>
</tr>
<tr>
<td>%[...]</td>
<td>Read the longest string containing characters specified in the brackets.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>%[^...]</td>
<td>Read the longest nonempty string containing characters that are not specified in the brackets.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>%*... instead of %</td>
<td>Ignore the matching characters specified by *.</td>
<td>No output</td>
</tr>
<tr>
<td>%w... instead of %</td>
<td>Read field width specified by w. The %f format supports %w.pf, where w is the field width and p is the precision.</td>
<td></td>
</tr>
</tbody>
</table>

[A,B,C,...] = textread('filename',format,N) reads the data, reusing the format string N times, where N is an integer greater than zero. If N is smaller than zero, textread reads the entire file.

[...] = textread(...,'param','value',...) customizes textread using param/value pairs, as listed in the table below.
<table>
<thead>
<tr>
<th>param</th>
<th>value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufsize</td>
<td>Positive integer</td>
<td>Specifies the maximum string length, in bytes. Default is 4095.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>matlab</td>
<td>Ignores characters after %.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>shell</td>
<td>Ignores characters after #.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>c</td>
<td>Ignores characters between /* and */.</td>
</tr>
<tr>
<td>commentstyle</td>
<td>c++</td>
<td>Ignores characters after //.</td>
</tr>
<tr>
<td>delimiter</td>
<td>One or more characters</td>
<td>Act as delimiters between elements. Default is none.</td>
</tr>
<tr>
<td>emptyvalue</td>
<td>Scalar double</td>
<td>Value given to empty cells when reading delimited files. Default is 0.</td>
</tr>
<tr>
<td>endofline</td>
<td>Single character or ' \r\n'</td>
<td>Character that denotes the end of a line. Default is determined from file.</td>
</tr>
<tr>
<td>expchars</td>
<td>Exponent characters</td>
<td>Default is eEdD.</td>
</tr>
<tr>
<td>headerlines</td>
<td>Positive integer</td>
<td>Ignores the specified number of lines at the beginning of the file.</td>
</tr>
<tr>
<td>whitespace</td>
<td>Any from the list below:</td>
<td>Treats vector of characters as white space. Default is ' \b\t'.</td>
</tr>
<tr>
<td></td>
<td>' '</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td></td>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td></td>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td></td>
<td>\t</td>
<td>Horizontal tab</td>
</tr>
</tbody>
</table>

**Note** When `textread` reads a consecutive series of `whitespace` values, it treats them as one white space. When it reads a consecutive series of `delimiter` values, it treats each as a separate delimiter.
Remarks
If you want to preserve leading and trailing spaces in a string, use the whitespace parameter as shown here:

```matlab
textread('myfile.txt', '%s', 'whitespace', '')
an =
   ' An example of preserving spaces '
```

Examples

**Example 1 — Read All Fields in Free Format File Using %**

The first line of `mydata.dat` is

Sally   Level1 12.34 45 Yes

Read the first line of the file as a free format file using the `%` format.

```matlab
[names, types, x, y, answer] = textread('mydata.dat', ...
   '%s %s %f %d %s', 1)
```

returns

```matlab
names =
   'Sally'
types =
   'Level1'
x =
   12.3400000000000000
y =
   45
answer =
   'Yes'
```

**Example 2 — Read as Fixed Format File, Ignoring the Floating Point Value**

The first line of `mydata.dat` is

Sally   Level1 12.34 45 Yes

Read the first line of the file as a fixed format file, ignoring the floating-point value.
[names, types, y, answer] = textread('mydata.dat', ...
  '%9c %5s %*f %2d %3s', 1)

returns

  names =
  Sally
  types =
      'Level1'
  y =
      45
  answer =
      'Yes'

%*f in the format string causes textread to ignore the floating point value, in this case, 12.34.

Example 3 — Read Using Literal to Ignore Matching Characters

The first line of mydata.dat is

    Sally    Type1  12.34  45  Yes

Read the first line of the file, ignoring the characters Type in the second field.

  [names, typenum, x, y, answer] = textread('mydata.dat', ...
    '%s Type%d %f %d %s', 1)

returns

  names =
      'Sally'
  typenum =
      1
  x =
      12.340000000000000
  y =
      45
answer = 'Yes'

Type%d in the format string causes the characters Type in the second
field to be ignored, while the rest of the second field is read as a signed
integer, in this case, 1.

**Example 4 — Specify Value to Fill Empty Cells**

For files with empty cells, use the emptyvalue parameter. Suppose
the file data.csv contains:

```
1,2,3,4,,6
7,8,9,,11,12
```

Read the file using NaN to fill any empty cells:

```matlab
data = textread('data.csv', '', 'delimiter', ',', ...
    'emptyvalue', NaN);
```

**Example 5 — Read M-File into a Cell Array of Strings**

Read the file fft.m into cell array of strings.

```matlab
file = textread('fft.m', '%s', 'delimiter', '
', ...
    'whitespace', '');
```

**See Also**
textscan, dlmread, csvread, fscanf
Purpose
Read formatted data from text file or string

Syntax
C = textscan(fid, 'format')
C = textscan(fid, 'format', N)
C = textscan(fid, 'format', param, value, ...)
C = textscan(fid, 'format', N, param, value, ...)
C = textscan(str, ...)
[C, position] = textscan(...)

Description
Note Before reading a file with `textscan`, you must open the file with the `fopen` function. `fopen` supplies the `fid` input required by `textscan`. When you are finished reading from the file, close the file by calling `fclose(fid)`.

C = textscan(fid, 'format') reads data from an open text file identified by file identifier `fid` into cell array `C`. The `format` input is a string of conversion specifiers enclosed in single quotation marks. The number of specifiers determines the number of cells in the cell array `C`.

C = textscan(fid, 'format', N) reads data from the file, using the `format` N times, where N is a positive integer. You can read additional data from the file after N cycles by calling `textscan` again using the original `fid`.

C = textscan(fid, 'format', param, value, ...) reads data from the file using nondefault parameter settings specified by one or more pairs of `param` and `value` arguments. For a list of all valid parameter strings, value descriptions, and defaults, see “User Configurable Options” on page 2-3716.

C = textscan(fid, 'format', N, param, value, ...) reads data from the file, using the `format` N times, and using nondefault parameter settings specified by pairs of `param` and `value` arguments.

C = textscan(str, ...) reads data from string `str`. You can use the `format`, `N`, and parameter/value arguments described above with this syntax. However, for strings, repeated calls to `textscan` restart the
scan from the beginning each time. (See “Example 10 — Resuming a Text Scan of a String” on page 2-3725.)

\[ [C, \text{position}] = \text{textscan}(\ldots) \]
returns the file or string position at the end of the scan as the second output argument. For a file, this is the value that \texttt{fseek}(\texttt{fid}) would return after calling \texttt{textscan}. For a string, \texttt{position} indicates how many characters \texttt{textscan} read.

**Remarks**

When \texttt{textscan} reads a specified file or string, it attempts to match the data to the format string. If \texttt{textscan} fails to convert a data field, it stops reading and returns all fields read before the failure.

**Basic Conversion Specifiers**

The format input is a string of one or more conversion specifiers. The following table lists the basic specifiers.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Specifier</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer, signed</td>
<td>%d</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td>%d8</td>
<td>8-bit</td>
</tr>
<tr>
<td></td>
<td>%d16</td>
<td>16-bit</td>
</tr>
<tr>
<td></td>
<td>%d32</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td>%d64</td>
<td>64-bit</td>
</tr>
<tr>
<td>Integer, unsigned</td>
<td>%u</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td>%u8</td>
<td>8-bit</td>
</tr>
<tr>
<td></td>
<td>%u16</td>
<td>16-bit</td>
</tr>
<tr>
<td></td>
<td>%u32</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td>%u64</td>
<td>64-bit</td>
</tr>
<tr>
<td>Floating-point number</td>
<td>%f</td>
<td>64-bit (double)</td>
</tr>
<tr>
<td></td>
<td>%f32</td>
<td>32-bit (single)</td>
</tr>
<tr>
<td></td>
<td>%f64</td>
<td>64-bit (double)</td>
</tr>
<tr>
<td></td>
<td>%n</td>
<td>64-bit (double)</td>
</tr>
<tr>
<td>Character strings</td>
<td>%s</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>%q</td>
<td>String, possibly double-quoted</td>
</tr>
<tr>
<td></td>
<td>%c</td>
<td>Any single character, including a delimiter</td>
</tr>
</tbody>
</table>
### Field Type Specifier Details

**Pattern-matching strings**

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Details</th>
</tr>
</thead>
</table>
| %[...]    | Read only characters in the brackets, until the first nonmatching character. Use %[...] to include ].  
Example: %[mus] reads 'summer' as 'summ'. |
| %[^...]   | Read only characters not in the brackets, until the first matching character. Use %[^]... to exclude ].  
Example: %[^xrg] reads 'summer' as 'summe'. |

For each numeric conversion specifier, `textscan` returns a K-by-1 MATLAB numeric vector to the output cell array `C`, where K is the number of times that `textscan` finds a field matching the specifier. For each string conversion specifier, `textscan` returns a K-by-1 cell vector of strings. For each character conversion of the form %Nc (see “Field Length” on page 2-3714), `textscan` returns a K-by-N character array.

### Field Length

You can specify the number of characters or digits to read by inserting a number between the percent character (%) and the format specifier. For floating-point numbers (%n, %f, %f32, %f64), you also can specify the number of digits read to the right of the decimal point.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Nc</td>
<td>Read N characters, including delimiter characters. Example: %9c reads 'Let's Go!' as 'Let's Go!'.</td>
</tr>
</tbody>
</table>
Specifier | Action Taken
--- | ---
%Ns %Nn | Read N characters or digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first.
%Nq %Nd... | Example: %5f32 reads '473.238' as 473.2.
%N[...] %Nu... | %N[^...]%Nf... |
%N.Dn | Read N digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Return D decimal digits in the output.
%N.Df... | Example: %7.2f reads '473.238' as 473.23.

**Skipping Fields or Parts of Fields**

The `textscan` function reads all characters in your file in sequence unless you tell it to ignore a particular field or a portion of a field.

Use the following format specifiers to skip or read portions of fields:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>%*...</td>
<td>Skip the field. <code>textscan</code> does not create an output cell for any field that it skips.</td>
</tr>
<tr>
<td></td>
<td>Example: '%s %*s %s %s %*s %s %s' (spaces are optional) converts the string 'Blackbird singing in the dead of night' to four output cells with the strings 'Blackbird' 'in' 'the' 'night'</td>
</tr>
</tbody>
</table>
**Specifier** | **Action Taken**
---|---
`%*n...` | Ignore \( n \) characters of the field, where \( n \) is an integer less than or equal to the number of characters in the field.  
Example: `%*4s` reads 'summer' as 'er'.

| literal | Ignore the specified characters of the field.  
Example: `Level%u8` reads 'Level1' as 1.  
Example: `%u8Step` reads '2Step' as 2. |

The `textscan` function does not include leading white-space characters in the processing of any data fields. When processing numeric data, `textscan` also ignores trailing white space.

**User Configurable Options**

This table shows the valid `param-value` options and their default values. Parameter names are not case sensitive.

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Value</strong></th>
<th><strong>Default</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BufSize</td>
<td>Maximum string length in bytes</td>
<td>4095</td>
</tr>
<tr>
<td>CollectOutput</td>
<td>If true, <code>textscan</code> concatenates consecutive output cells with the same data type into a single array.</td>
<td>0 (false)</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Default</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CommentStyle</td>
<td>Symbol(s) designating text to ignore. Specify a single string (such as '%') to ignore characters following the string on the same line. Specify a cell array of two strings (such as {'/<em>', '</em>/*'}) to ignore characters between the strings.</td>
<td>None</td>
</tr>
<tr>
<td>Delimiter</td>
<td>Field delimiter character(s)</td>
<td>White space</td>
</tr>
<tr>
<td>EmptyValue</td>
<td>Value to return for empty numeric fields in delimited files</td>
<td>NaN</td>
</tr>
<tr>
<td>EndOfLine</td>
<td>End-of-line character</td>
<td>Determined from the file: \n, \r, or \r\n</td>
</tr>
<tr>
<td>ExpChars</td>
<td>Exponent characters</td>
<td>'eEdD'</td>
</tr>
<tr>
<td>HeaderLines</td>
<td>Number of lines to skip. (Includes the remainder of the current line.)</td>
<td>0</td>
</tr>
<tr>
<td>MultipleDelimsAsOne</td>
<td>If true, textscan treats consecutive delimiters as a single delimiter. Only valid if you specify the delimiter option.</td>
<td>0 (false)</td>
</tr>
</tbody>
</table>
## Parameter Value Default

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReturnOnError</td>
<td>Determines behavior when <code>textscan</code> fails to read or convert. If true, <code>textscan</code> terminates without an error and returns all fields read. If false, <code>textscan</code> terminates with an error and does not return an output cell array.</td>
<td>1 (true)</td>
</tr>
<tr>
<td>TreatAsEmpty</td>
<td>String(s) in the data file to treat as an empty value. Can be a single string or cell array of strings. Only applies to numeric fields.</td>
<td>None</td>
</tr>
<tr>
<td>Whitespace</td>
<td>White-space characters</td>
<td>' \b\t'</td>
</tr>
</tbody>
</table>

### Field and Row Delimiters

Within each row, the default field delimiter is white space. White space can be any combination of space (' '), backspace (' \b'), or tab (' \t') characters.

If you use the default (white space) field delimiter, `textscan` interprets repeated white-space characters as a single delimiter. If you specify a nondefault delimiter, `textscan` interprets repeated delimiter characters as separate delimiters, and returns an empty value to the output cell. (See “Example 5 — Specifying Delimiter and Empty Value Conversion” on page 2-3721 and “Example 7 — Handling Repeated Delimiters” on page 2-3722.)

Rows delimiters are end-of-line (EOL) character sequences. The default end-of-line setting depends on the format of your file, and can include a newline character (' \n'), a carriage return (' \r'), or a combination of the two (' \r\n').
For more information, see “Example 9 — Using Nondefault Control Characters” on page 2-3724.

**Numeric Fields**

textscan converts numeric fields to the specified output type according to MATLAB rules regarding overflow, truncation, and the use of NaN, Inf, and -Inf.

For example, MATLAB represents an integer NaN as zero. If textscan finds an empty field associated with an integer format specifier (such as %d or %u), it returns the empty value as zero and not NaN. (See “Example 2 — Reading Different Types of Data” on page 2-3720 and “Example 5 — Specifying Delimiter and Empty Value Conversion” on page 2-3721.)

textscan imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. Valid forms for a complex number are as follows:

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>±&lt;real&gt;±&lt;imag&gt;i</td>
<td>5.7-3.1i</td>
</tr>
<tr>
<td>±&lt;imag&gt;i</td>
<td>-7j</td>
</tr>
</tbody>
</table>

Do not include embedded white space in a complex number. textscan interprets embedded white space as a field delimiter.

**Examples**

**Note** The following examples include spaces between the conversion specifiers to make the format value easier to read. Spaces are not required.

**Example 1 — Reading a String**

Read the following string, truncating each value to one decimal digit. The specifier %*1d tells textscan to skip the remaining digit:

```matlab
str = '0.41 8.24 3.57 6.24 9.27';
```
textscan

C = textscan(str, '%3.1f %*1d');

textscan returns a 1-by-1 cell array C:

C{1} = [0.4; 8.2; 3.5; 6.2; 9.2]

Example 2 — Reading Different Types of Data

The text file scan1.dat contains data in the following form:

Sally Level1 12.34 45 1.23e10 inf NaN Yes
Joe Level2 23.54 60 9e19 -inf 0.001 No
Bill Level3 34.90 12 2e5 10 100 No

Open the file, and read each column with the appropriate conversion specifier:

fid = fopen('scan1.dat');
C = textscan(fid, '%s %s %f32 %d8 %u %f %f %s');
fclose(fid);

textscan returns a 1-by-8 cell array C with the following cells:

C{1} = {'Sally'; 'Joe'; 'Bill'} class cell
C{2} = {'Level1'; 'Level2'; 'Level3'} class cell
C{3} = [12.34; 23.54; 34.9] class single
C{4} = [45; 60; 12] class int8
C{5} = [4294967295; 4294967295; 200000] class uint32
C{6} = [Inf; -Inf; 10] class double
C{7} = [NaN; 0.001; 100] class double
C{8} = {'Yes'; 'No'; 'No'} class cell

The first two elements of C{5} are the maximum values for a 32-bit unsigned integer, or intmax('uint32').

Example 3 — Removing a Literal String

Remove the text 'Level' from each field in the second column of the data from Example 2:

fid = fopen('scan1.dat');
C = textscan(fid, '%s Level%u8 %f32 %d8 %u %f %f %s');
fclose(fid);

textscan returns a 1-by-8 cell array, C, with

\[
C\{2\} = \begin{bmatrix} 1; 2; 3 \end{bmatrix} \quad \text{class} \quad \text{uint8}
\]

**Example 4 — Reading Only the First Field**

Read the first column of the file in Example 2 into a cell array, skipping
the rest of the line:

\[
\begin{align*}
\text{fid} &= \text{fopen('scan1.dat')} ; \\
\text{names} &= \text{textscan}\left(\text{fid, '%s %*[^[n]}'\right) ; \\
\text{fclose}\left(\text{fid}\right)
\end{align*}
\]

textscan returns a 1-by-1 cell array names:

\[
\text{names}\{1\} = \{ 'Sally'; 'Joe'; 'Bill' \}
\]

**Example 5 — Specifying Delimiter and Empty Value Conversion**

The comma-delimited file data.csv contains

\[
\begin{align*}
1, & \quad 2, \quad 3, \quad 4, \quad , \quad 6 \\
7, & \quad 8, \quad 9, \quad , \quad 11, \quad 12
\end{align*}
\]

Read the file, converting empty cells to -Inf:

\[
\begin{align*}
\text{fid} &= \text{fopen('data.csv')}; \\
C &= \text{textscan}\left(\text{fid, '%f %f %f %f %u32 %f', 'delimiter', ',', 'EmptyValue', -Inf} \right) ; \\
\text{fclose}\left(\text{fid}\right)
\end{align*}
\]

textscan returns a 1-by-6 cell array C with the following cells:

\[
\begin{align*}
C\{1\} &= \begin{bmatrix} 1; 7 \end{bmatrix} \quad \text{class} \quad \text{double} \\
C\{2\} &= \begin{bmatrix} 2; 8 \end{bmatrix} \quad \text{class} \quad \text{double} \\
C\{3\} &= \begin{bmatrix} 3; 9 \end{bmatrix} \quad \text{class} \quad \text{double}
\end{align*}
\]
C{4} = [4; -Inf] class double (empty converted to -Inf)
C{5} = [0; 11] class uint32 (empty converted to 0)
C{6} = [6; 12] class double

textscan converts the empty value in C{4}, associated with a floating-point format, to -Inf. Because MATLAB represents unsigned integer -Inf as 0, textscan converts the empty value in C{5} to 0 and not -Inf.

**Example 6 — Using Custom Empty Value Strings and Comments**

The comma-delimited file data2.csv contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```

Designate the input that textscan should treat as comments or empty values:

```matlab
fid = fopen('data2.csv');
C = textscan(fid, '%s %n %n %n %n', 'delimiter', ',', ...
    'treatAsEmpty', {'NA', 'na'}, ...
    'commentStyle', '//');
fclose(fid);
```

textscan returns a 1-by-5 cell array C with the following cells:

- C{1} = {'abc'; 'def'}
- C{2} = [2; NaN]
- C{3} = [NaN; 5]
- C{4} = [3; 6]
- C{5} = [4; 7]

**Example 7 — Handling Repeated Delimiters**

The file data3.csv contains

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

To treat the repeated commas as a single delimiter, use the MultipleDelimsAsOne parameter, with a value of 1:

```matlab
fid = fopen('data3.csv');
C = textscan(fid, '%f %f %f %f', 'delimiter', ',', ...
    'MultipleDelimsAsOne', 1);
fclose(fid);
```

textscan returns a 1-by-4 cell array C with the following cells:

- `C{1}` = [1; 5]
- `C{2}` = [2; 6]
- `C{3}` = [3; 7]
- `C{4}` = [4; 8]

**Example 8 — Using the CollectOutput Switch**

The file `grades.txt` contains

<table>
<thead>
<tr>
<th>Student_ID</th>
<th>Test1</th>
<th>Test2</th>
<th>Test3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>91.5</td>
<td>89.2</td>
<td>77.3</td>
</tr>
<tr>
<td>2</td>
<td>88.0</td>
<td>67.8</td>
<td>91.0</td>
</tr>
<tr>
<td>3</td>
<td>76.3</td>
<td>78.1</td>
<td>92.5</td>
</tr>
<tr>
<td>4</td>
<td>96.4</td>
<td>81.2</td>
<td>84.6</td>
</tr>
</tbody>
</table>

The default value for the CollectOutput switch is 0 (false), and textscan returns each column of the numeric data in a separate array:

```matlab
fid = fopen('grades.txt');

% read column headers
C_text = textscan(fid, '%s', 4, 'delimiter', '|');

% read numeric data
C_data0 = textscan(fid, '%d %f %f %f')

C_data0 =
```

2-3723
Textscan

Set CollectOutput to 1 (true) to collect the consecutive columns of the same class (the test scores, which are all double) into a single array:

```matlab
rewind(fid);
C_text = textscan(fid, '%s', 4, 'delimiter', '|');
C_data1 = textscan(fid, '%d %f %f %f', ...
    'CollectOutput', 1)
C_data1 =
    [4x1 int32]  [4x3 double]
fclose(fid);
```

Example 9 — Using Nondefault Control Characters

When you specify one of the following escape sequences for any parameter value, textscan converts that sequence to the corresponding control character:

```
\b  Backspace
\n  Newline
\r  Carriage return
\t  Tab
\\  Backslash (\)
```

If your data uses a different control character, use the sprintf function to explicitly convert the escape sequence in your call to textscan.

For example, the following string includes a form feed character, \f:

```matlab
lyric = sprintf('Blackbird\fsinging\fin\fthe\fdead\fof\fnight');
```
To read the string using `textscan`, call the `sprintf` function to explicitly convert the form feed:

```matlab
C = textscan(lyric, '%s', 'delimiter', sprintf(''));
```

textscan returns a 1-by-1 cell array `C`:

```matlab
C{1} =
    {'Blackbird'; 'singing'; 'in'; 'the'; 'dead'; 'of'; 'night'}
```

**Example 10 — Resuming a Text Scan of a String**

If you resume a text scan of a file by calling `textscan` with the same file identifier (fid), `textscan` automatically resumes reading at the point where it terminated the last read.

If your input is a string rather than a file, `textscan` reads from the beginning of the string each time. To resume a scan from any other position in the string, you must use the two-output argument syntax in your initial call to `textscan`. For example, given the string

```matlab
lyric = 'Blackbird singing in the dead of night'
```

Read the first word of the string:

```matlab
[firstword, pos] = textscan(lyric, '%9c', 1);
```

Resume the scan:

```matlab
lastpart = textscan(lyric(pos+1:end), '%s');
```

**See Also**

`load`, `type`, `importdata`, `uiimport`, `dlmread`, `xlsread`, `fscanf`, `fread`

"Importing Large Data Sets" in the MATLAB Programming Fundamentals documentation
Wrapped string matrix for given uicontrol

```matlab
outstring = textwrap(h,instring)
[outstring,position]=textwrap(h,instring)
```

returns a wrapped string cell array, `outstring`, that fits inside the uicontrol with handle `h`. `instring` is a cell array, with each cell containing a single line of text. `outstring` is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.

```matlab
[outstring,position]=textwrap(h,instring)
```

returns the recommended position of the uicontrol in the units of the uicontrol. `position` considers the extent of the multiline text in the `x` and `y` directions.

When programming a GUI, do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object’s `RecursionLimit` property.

Place a text-wrapped string in a uicontrol:

```matlab
pos = [10 10 100 10];
h = uicontrol('Style','Text','Position',pos);
string = {'This is a string for the uicontrol.',
          'It should be correctly wrapped inside.'};
[outstring,newpos] = textwrap(h,string);
pos(4) = newpos(4);
set(h,'String',outstring,'Position',[pos(1),pos(2),pos(3)+10,pos(4)])
```

See Also

`uicontrol`
### Purpose
Transpose-free quasi-minimal residual method

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = tfqmr(A,b)</code></td>
<td>Attempts to solve the system of linear equations $A*x=b$ for $x$. The $n$-by-$n$ coefficient matrix $A$ must be square and the right-hand side column vector $b$ must have length $n$.</td>
</tr>
<tr>
<td><code>x = tfqmr(afun,b)</code></td>
<td>Accepts a function handle <code>afun</code> instead of the matrix $A$. <code>afun(x)</code> accepts a vector input $x$ and returns the matrix-vector product $A*x$. In all of the following syntaxes, you can replace $A$ by <code>afun</code>. See “Function Handles” in the MATLAB Programming documentation for more information. “Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function <code>afun</code>.</td>
</tr>
<tr>
<td><code>x = tfqmr(a,b,tol)</code></td>
<td>Specifies the tolerance of the method. If <code>tol</code> is [] then <code>tfqmr</code> uses the default, $1e-6$.</td>
</tr>
<tr>
<td><code>x = tfqmr(a,b,tol,maxit)</code></td>
<td>Specifies the maximum number of iterations. If <code>maxit</code> is [] then <code>tfqmr</code> uses the default, $\min(N,20)$.</td>
</tr>
<tr>
<td><code>x = tfqmr(a,b,tol,maxit,m)</code> and <code>x = tfqmr(a,b,tol,maxit,m1,m2)</code></td>
<td>Use preconditioners $m$ or $m = m1<em>m2$ and effectively solve the system $A</em>\text{inv}(M)*x = B$ for $x$. If $M$ is [] then a preconditioner is not applied. $M$ may be a function handle <code>mfun</code> such that <code>mfun(x)</code> returns $m\backslash x$.</td>
</tr>
<tr>
<td><code>x = tfqmr(a,b,tol,maxit,m1,m2,x0)</code></td>
<td>Specifies the initial guess. If <code>x0</code> is [] then <code>tfqmr</code> uses the default, an all zero vector.</td>
</tr>
</tbody>
</table>
[x,flag] = tfqmr(A,B,...) also returns a convergence flag:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>tfqmr converged to the desired tolerance tol within maxit iterations.</td>
</tr>
<tr>
<td>1</td>
<td>tfqmr iterated maxit times but did not converge.</td>
</tr>
<tr>
<td>2</td>
<td>Preconditioner m was ill-conditioned.</td>
</tr>
<tr>
<td>3</td>
<td>tfqmr stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during tfqmr became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

[x,flag,relres] = tfqmr(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, then relres <= tol.

[x,flag,relres,y] = tfqmr(A,b,...) also returns the iteration number at which x was computed: 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = tfqmr(A,b,...) also returns a vector of the residual norms at each iteration, including norm(b-A*x0).

### Examples

```matlab
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = tfqmr(A,b,tol,maxit,M1,M2,[]);
```

You can also use a matrix-vector product function as input:

```matlab
function y = afun(x,n)
y = 4 * x;
y(2:n) = y(2:n) - 2 * x(1:n-1);
```
\[ y(1:n-1) = y(1:n-1) - x(2:n); \]
\[ x1 = tfqmr(@(x)afun(x,n),b,tol,maxit,M1,M2); \]

If \texttt{applyOp} is a function suitable for use with \texttt{qmr}, it may be used with \texttt{tfqmr} by wrapping it in an anonymous function:

\[ x1 = tfqmr(@(x)applyOp(x,'notransp'),b,tol,maxit,M1,M2); \]

\textbf{See Also} \hspace{1cm} \texttt{qmr, bicg, bicgstab, bicgstablecgs, gmres, lsqr, luinc, minres, pcg, symmlq, mldivide}
throw (MException)

Purpose
Terminate function and issue exception

Syntax
throw(ME)

Description
throw(ME) terminates the currently running function, issues an exception based on MException object ME, and returns control to the keyboard or to any enclosing catch block. A thrown MException displays a message in the Command Window unless it is caught by try-catch. throw also sets the MException stack field to the location from which the throw method was called.

Examples

Example 1
This example tests the output of M-file evaluate_plots and throws an exception if it is not acceptable:

```
[minval, maxval] = evaluate_plots(p24, p28, p41);
if minval < lower_bound || maxval > upper_bound
    ME = MException('VerifyOutput:OutOfBounds',...
                    'Results are outside the allowable limits');
    throw(ME);
end
```

Example 2
This example attempts to open a file in a directory that is not on the MATLAB path. It uses a nested try-catch block to give the user the opportunity to extend the path. If the file still cannot be found, the program issues an exception with the first error appended to the second:

```
function data = read_it(filename);
try
    fid = fopen(filename, 'r');
    data = fread(fid);
catch eObj1
    if strcmp(eObj1.identifier, 'MATLAB:FileIO:InvalidFid')
        msg = sprintf('
%s%s%s', 'Cannot open file ', ...
                      filename, '. Try another location? ');
    end
```
reply = input(msg, 's')
if reply(1) == 'y'
    newdir = input('Enter directory name: ', 's');
else
    throw(eObj1);
end
addpath(newdir);
try
    fid = fopen(filename, 'r');
    data = fread(fid);
catch eObj2
    eObj3 = addCause(eObj2, eObj1)
    throw(eObj3);
end
rmpath(newdir);
end
fclose(fid);

If you run this function in a try-catch block at the command line, you can look at the MException object by assigning it to a variable (e) with the catch command.

try
    d = read_it('anytextfile.txt');
catch e
end

e = MException object with properties:

    identifier: 'MATLAB:FileIO:InvalidFid'
    message: 'Invalid file identifier. Use fopen to generate a valid file identifier.'
    stack: [1x1 struct]
    cause: {{[1x1 MException]}}
throw (MException)

Cannot open file anytextfile.txt. Try another location? y
Enter directory name: xxxxxxx
Warning: Name is nonexistent or not a directory: xxxxxxx.
> In path at 110
  In addpath at 89

See Also

error, try, catch, assert, MException, rethrow(MException),
throwAsCaller(MException), addCause(MException),
getReport(MException), disp(MException), isequal(MException),
eq(MException), ne(MException), last(MException),
**Throw exception, as if from calling function**

**Syntax**

`throwAsCaller(ME)`

**Description**

`throwAsCaller(ME)` throws an exception from the currently running M-file based on `MException` object `ME`. The MATLAB software exits the currently running function and returns control to either the keyboard or an enclosing `catch` block in a calling function. Unlike the `throw` function, MATLAB omits the current stack frame from the `stack` field of the `MException`, thus making the exception look as if it is being thrown by the caller of the function.

In some cases, it is not relevant to show the person running your program the true location that generated an exception, but is better to point to the calling function where the problem really lies. You might also find `throwAsCaller` useful when you want to simplify the error display, or when you have code that you do not want made public.

**Examples**

The function `klein_bottle`, in this example, generates a Klein Bottle figure by revolving the figure-eight curve defined by `XYKLEIN`. It defines a few variables and calls the function `draw_klein`, which executes three functions in a `try-catch` block. If there is an error, the `catch` block issues an exception using either `throw` or `throwAsCaller`:

```matlab
function klein_bottle(ab, pq)
    rtr = [2 0.5 1];
    box = [-3 3 -3 3 -2 2];
    vue = [55 60];
    draw_klein(ab, rtr, pq, box, vue)
end

function draw_klein(ab, rtr, pq, box, vue)
    clf
    try
        tube('xyklein',ab, rtr, pq, box, vue);
        shading interp
        colormap(pink);
    end
end
```
Call the `klein_bottle` function, passing an incorrect value for the second argument. (The correct value would be a vector, such as `[40 40].`) Because the catch block issues the exception using `throw`, MATLAB displays error messages for line 15 of function `draw_klein`, and for line 5 of function `klein_bottle`:

```
klein_bottle(ab, pi)
??? Attempted to access pq(2); index out of bounds because numel(pq)=1.
```

```
Error in ==> klein_bottle>draw_klein at 15
throw(ME);
```

```
Error in ==> klein_bottle at 5
draw_figure(ab, rtr, pq, box, vue)
```

Run the function again, this time changing the `klein_bottle.m` file so that the catch block uses `throwAsCaller` instead of `throw`. This time, MATLAB only displays the error at line 5 of the main program:

```
klein_bottle(ab, pi)
??? Attempted to access pq(2); index out of bounds because numel(pq)=1.
```

```
Error in ==> klein_bottle at 5
draw_figure(ab, rtr, pq, box, vue)
```

**See Also**

`error`, `try`, `catch`, `assert`, `MException`, `throw(MException)`,
`rethrow(MException)`, `addCause(MException)`,
`getReport(MException)`, `disp(MException)`, `isequal(MException)`,
`eq(MException)`, `ne(MException)`, `last(MException)`
**Purpose**  
Measure performance using stopwatch timer

**Syntax**  
```
tic; any_statements; toc;
tic; any_statements; tElapsed=toc;
tStart=tic; any_statements; toc(tStart);
tStart=tic; any_statements; tElapsed=toc(tStart);
```

**Description**  
```
tic; any_statements; toc; measures the time it takes the MATLAB software to execute the one or more lines of MATLAB code shown here as any_statements. The tic command starts a stopwatch timer, MATLAB executes the block of statements, and toc stops the timer, displaying the time elapsed in seconds.
```
```
tic; any_statements; tElapsed=toc; makes the same time measurement, but assigns the elapsed time output to a variable, tElapsed. MATLAB does not display the elapsed time unless you omit the terminating semicolon. The value returned by toc is a scalar double that represents the elapsed time in seconds.
```
```
tStart=tic; any_statements; toc(tStart); makes the same time measurement, but allows you the option of running more than one stopwatch timer concurrently. You assign the output of tic to a variable tStart and then use that same variable when calling toc. MATLAB measures the time elapsed between the tic and its related toc command and displays the time elapsed in seconds. This syntax enables you to time multiple concurrent operations, including the timing of nested operations.
```
```
tStart=tic; any_statements; tElapsed=toc(tStart); is the same as the command shown above, except that MATLAB assigns the elapsed time output to a variable, tElapsed. MATLAB does not display the elapsed time unless you omit the terminating semicolon. The value returned by toc is a scalar double that represents the elapsed time in seconds.
```

**Remarks**  
Using the third syntax shown above, you can nest tic-toc pairs.
When using the simpler tic and toc syntax, avoid using consecutive tics as they merely overwrite the internally-recorded starting time. Consecutive tocs however, may be useful as each toc returns the increasing time that has elapsed since the most recent tic. Using this mechanism, you can take multiple measurements from a single point in time.

When using the tStart=tic and toc(tStart) syntax, it is advisable to select a unique variable for tStart. If you accidentally overwrite this variable prior to the toc for which it is needed, you will get inaccurate results for the time measurement.

tStart is a 64-bit unsigned integer, scalar value. This value is only useful as an input argument for a subsequent call to toc.

The clear function does not reset the starting time recorded by a tic command.

**Examples**

Measure how the time required to solve a linear system varies with the order of a matrix:

```matlab
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic
    x = A\b;
    t(n) = toc;
end
plot(t)
```

Measure the minimum and average time to compute a summation of Bessel functions:

```matlab
REPS = 1000;  minTime = Inf;  nsum = 10;
tic;

for i=1:REPS
    tStart = tic;  total = 0;
    for j=1:nsum,
```
total = total + besselj(j,REPS);
end

tElapsed = toc(tStart);
minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;

See Also    clock, cputime, etime, profile
Purpose

Construct timer object

Syntax

T = timer
T = timer('PropertyName1', PropertyValue1, 'PropertyName2', PropertyValue2, ...)

Description

T = timer constructs a timer object with default attributes.

T = timer('PropertyName1', PropertyValue1, 'PropertyName2', PropertyValue2, ...) constructs a timer object in which the given property name/value pairs are set on the object. See “Timer Object Properties” on page 2-3738 for a list of all the properties supported by the timer object.

Note that the property name/property value pairs can be in any format supported by the set function, i.e., property/value string pairs, structures, and property/value cell array pairs.

Examples

This example constructs a timer object with a timer callback function handle, mycallback, and a 10 second interval.

    t = timer('TimerFcn',@mycallback, 'Period', 10.0);

See Also

delete(timer), disp(timer), get(timer), isvalid(timer), set(timer), start, startat, stop, timerfind, timerfindall, wait

Timer Object Properties

The timer object supports the following properties that control its attributes. The table includes information about the data type of each property and its default value.

To view the value of the properties of a particular timer object, use the get(timer) function. To set the value of the properties of a timer object, use the set(timer) function.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Data Types, Values, Defaults, Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AveragePeriod</strong></td>
<td>Average time between TimerFcn executions since the timer started. Note: Value is NaN until timer executes two timer callbacks.</td>
<td>Data type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only</td>
</tr>
</tbody>
</table>
| **BusyMode** | Action taken when a timer has to execute TimerFcn before the completion of previous execution of TimerFcn.  

- 'drop' — Do not execute the function.  
- 'error' — Generate an error. Requires ErrorFcn to be set.  
- 'queue' — Execute function at next opportunity. | Data type | Enumerated string |
|               |                      | Values | 'drop'  
|               |                      |        | 'error'  
<p>|               |                      |        | 'queue' |
|               |                      | Default | 'drop' |
|               |                      | Read only | While Running = 'on' |
| <strong>ErrorFcn</strong> | Function that the timer executes when an error occurs. This function executes before the StopFcn. See “Creating Callback Functions” for more information. | Data type | Text string, function handle, or cell array |
|               |                      | Default | None |
|               |                      | Read only | Never |</p>
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Data Types, Values, Defaults, Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutionMode</td>
<td>Determines how the timer object schedules timer events. See “Timer Object Execution Modes” for more information.</td>
<td>Data type: Enumerated string, Values: 'singleShot', 'fixedDelay', 'fixedRate', 'fixedSpacing', Default: 'singleShot', Read only: While Running = 'on'</td>
</tr>
<tr>
<td>InstantPeriod</td>
<td>The time between the last two executions of TimerFcn.</td>
<td>Data type: double, Default: NaN, Read only: Always</td>
</tr>
<tr>
<td>Name</td>
<td>User-supplied name.</td>
<td>Data type: Text string, Default: 'timer-i', where i is a number indicating the ith timer object created this session. To reset i to 1, execute the clear classes command, Read only: Never</td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Data Types, Values, Defaults, Access</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>ObjectVisibility</td>
<td>Provides a way for application developers to prevent end-user access to the timer objects created by their application. The timerfind function does not return an object whose ObjectVisibility property is set to 'off'. Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that created it), you can set its properties.</td>
<td>Data type: Enumerated string Values: 'off', 'on' Default: 'on' Read only: Never</td>
</tr>
<tr>
<td>Period</td>
<td>Specifies the delay, in seconds, between executions of TimerFcn.</td>
<td>Data type: double Value: Any number &gt;= 0.001 Default: 1.0 Read only: While Running = 'on'</td>
</tr>
<tr>
<td>Running</td>
<td>Indicates whether the timer is currently executing.</td>
<td>Data type: Enumerated string Values: 'off', 'on' Default: 'off' Read only: Always</td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Data Types, Values, Defaults, Access</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>StartDelay</td>
<td>Specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in TimerFcn.</td>
<td>Data type: double Values: Any number &gt;= 0 Default: 0 Read only: While Running = 'on'</td>
</tr>
<tr>
<td>StartFcn</td>
<td>Function the timer calls when it starts. See “Creating Callback Functions” for more information.</td>
<td>Data type: Text string, function handle, or cell array Default: None Read only: Never</td>
</tr>
<tr>
<td><strong>Property Name</strong></td>
<td><strong>Property Description</strong></td>
<td><strong>Data Types, Values, Defaults, Access</strong></td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>StopFcnn</td>
<td>Function the timer calls when it stops. The timer stops when</td>
<td>Date type Text string, function handle, or cell array</td>
</tr>
<tr>
<td></td>
<td>- You call the timer stop function</td>
<td>Default None</td>
</tr>
<tr>
<td></td>
<td>- The timer finishes executing TimerFcn, i.e., the value of TasksExecuted reaches the limit set by TasksToExecute.</td>
<td>Read only Never</td>
</tr>
<tr>
<td></td>
<td>- An error occurs (The ErrorFcn is called first, followed by the StopFcn.)</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>User supplied label.</td>
<td>Data type Text string</td>
</tr>
<tr>
<td></td>
<td>Default Empty string (&quot;&quot;&quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read only Never</td>
<td></td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Data Types, Values, Defaults, Access</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td><strong>TasksToExecute</strong></td>
<td>Specifies the number of times the timer should execute the function specified in the TimerFcn property.</td>
<td>Data type double, Values Any number &gt; 0, Default 1, Read only Never</td>
</tr>
<tr>
<td><strong>TasksExecuted</strong></td>
<td>The number of times the timer has called TimerFcn since the timer was started.</td>
<td>Data type double, Values Any number &gt;= 0, Default 0, Read only Always</td>
</tr>
<tr>
<td><strong>TimerFcn</strong></td>
<td>Timer callback function. See “Creating Callback Functions” for more information.</td>
<td>Data type Text string, function handle, or cell array, Default None, Read only Never</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Identifies the object type.</td>
<td>Data type Text string, Values 'timer', Read only Always</td>
</tr>
<tr>
<td><strong>UserData</strong></td>
<td>User-supplied data.</td>
<td>Data type User-defined, Default [], Read only Never</td>
</tr>
</tbody>
</table>
Purpose

Find timer objects

Syntax

\[
\text{out} = \text{timerfind} \\
\text{out} = \text{timerfind('P1', V1, 'P2', V2,...)} \\
\text{out} = \text{timerfind(S)} \\
\text{out} = \text{timerfind(obj, 'P1', V1, 'P2', V2,...)}
\]

Description

\[
\text{out} = \text{timerfind} \text{ returns an array, out, of all the timer objects that exist in memory.}
\]

\[
\text{out} = \text{timerfind('P1', V1, 'P2', V2,...) returns an array, out, of timer objects whose property values match those passed as parameter/value pairs, P1, V1, P2, V2. Parameter/value pairs may be specified as a cell array.}
\]

\[
\text{out} = \text{timerfind(S) returns an array, out, of timer objects whose property values match those defined in the structure, S. The field names of S are timer object property names and the field values are the corresponding property values.}
\]

\[
\text{out} = \text{timerfind(obj, 'P1', V1, 'P2', V2,...) restricts the search for matching parameter/value pairs to the timer objects listed in obj. obj can be an array of timer objects.}
\]

Note

When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to timerfind.

Note that, for most properties, timerfind performs case-sensitive searches of property values. For example, if the value of an object’s Name property is 'MyObject', timerfind will not find a match if you specify 'myobject'. Use the get function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, timerfind will find an object with an ExecutionMode property value of 'singleShot' or 'singleshot'.

2-3745
Examples

These examples use `timerfind` to find timer objects with the specified property values.

```matlab
  t1 = timer('Tag', 'broadcastProgress', 'Period', 5);
  t2 = timer('Tag', 'displayProgress');
  out1 = timerfind('Tag', 'displayProgress')
  out2 = timerfind({'Period', 'Tag'}, {5, 'broadcastProgress'})
```

See Also

`get(timer)`, `timer`, `timerfindall`
Purpose

Find timer objects, including invisible objects

Syntax

out = timerfindall
out = timerfindall('P1', V1, 'P2', V2,...)
out = timerfindall(S)
out = timerfindall(obj, 'P1', V1, 'P2', V2,...)

Description

out = timerfindall returns an array, out, containing all the timer objects that exist in memory, regardless of the value of the object's ObjectVisibility property.

out = timerfindall('P1', V1, 'P2', V2,...) returns an array, out, of timer objects whose property values match those passed as parameter/value pairs, P1, V1, P2, V2. Parameter/value pairs may be specified as a cell array.

out = timerfindall(S) returns an array, out, of timer objects whose property values match those defined in the structure, S. The field names of S are timer object property names and the field values are the corresponding property values.

out = timerfindall(obj, 'P1', V1, 'P2', V2,...) restricts the search for matching parameter/value pairs to the timer objects listed in obj. obj can be an array of timer objects.

Note

When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to timerfindall.

Note that, for most properties, timerfindall performs case-sensitive searches of property values. For example, if the value of an object's Name property is 'MyObject', timerfindall will not find a match if you specify 'myobject'. Use the get function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, timerfindall will find an object with an ExecutionMode property value of 'singleShot' or 'singleshot'.
Examples

Create several timer objects.

```matlab
t1 = timer;
t2 = timer;
t3 = timer;
```

Set the `ObjectVisibility` property of one of the objects to 'off'.

```matlab
t2.ObjectVisibility = 'off';
```

Use `timerfind` to get a listing of all the timer objects in memory. Note that the listing does not include the timer object (timer-2) whose `ObjectVisibility` property is set to 'off'.

```matlab
timerfind
```

**Timer Object Array**

<table>
<thead>
<tr>
<th>Index</th>
<th>ExecutionMode</th>
<th>Period</th>
<th>TimerFcn</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>singleShot</td>
<td>1</td>
<td>''</td>
<td>timer-1</td>
</tr>
<tr>
<td>2</td>
<td>singleShot</td>
<td>1</td>
<td>''</td>
<td>timer-3</td>
</tr>
</tbody>
</table>

Use `timerfindall` to get a listing of all the timer objects in memory. This listing includes the timer object whose `ObjectVisibility` property is set to 'off'.

```matlab
timerfindall
```

**Timer Object Array**

<table>
<thead>
<tr>
<th>Index</th>
<th>ExecutionMode</th>
<th>Period</th>
<th>TimerFcn</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>singleShot</td>
<td>1</td>
<td>''</td>
<td>timer-1</td>
</tr>
<tr>
<td>2</td>
<td>singleShot</td>
<td>1</td>
<td>''</td>
<td>timer-2</td>
</tr>
<tr>
<td>3</td>
<td>singleShot</td>
<td>1</td>
<td>''</td>
<td>timer-3</td>
</tr>
</tbody>
</table>

See Also

`get(timer), timer, timerfind`
Purpose

Create timeseries object

Syntax

\[
\text{ts} = \text{timeseries} \\
\text{ts} = \text{timeseries}(\text{Data}) \\
\text{ts} = \text{timeseries}(\text{Name}) \\
\text{ts} = \text{timeseries}(\text{Data},\text{Time}) \\
\text{ts} = \text{timeseries}(\text{Data},\text{Time},\text{Quality}) \\
\text{ts} = \text{timeseries}(\text{Data},...,\text{Parameter},\text{Value},...) \\
\]

Description

\text{ts} = \text{timeseries} \text{ creates an empty time-series object.}

\text{ts} = \text{timeseries}(\text{Data}) \text{ creates a time series with the specified Data.}
\text{ts has a default time vector that ranges from 0 to N-1 with a 1-second interval, where N is the number of samples. The default name of the timeseries object is 'unnamed'.}

\text{ts} = \text{timeseries}(\text{Name}) \text{ creates an empty time series with the name specified by a string Name. This name can differ from the time-series variable name.}

\text{ts} = \text{timeseries}(\text{Data},\text{Time}) \text{ creates a time series with the specified Data array and Time. When time values are date strings, you must specify Time as a cell array of date strings.}

\text{ts} = \text{timeseries}(\text{Data},\text{Time},\text{Quality}) \text{ creates a timeseries object. The Quality attribute is an integer vector with values -128 to 127 that specifies the quality in terms of codes defined by QualityInfo.Code.}

\text{ts} = \text{timeseries}(\text{Data},...,\text{Parameter},\text{Value},...) \text{ creates a timeseries object with optional parameter-value pairs after the Data, Time, and Quality arguments. You can specify the following parameters:}

- \text{Name} — Time-series name entered as a string
- \text{IsTimeFirst} — Logical value (true or false) specifying whether the first or last dimension of the data array is aligned with the time vector. You can set this property when the data array is square and, therefore, the dimension that is aligned with time is ambiguous.
Remarks

Definition: timeseries

The time-series object, called timeseries, is a MATLAB variable that contains time-indexed data and properties in a single, coherent structure. For example, in addition to data and time values, you can also use the time-series object to store events, descriptive information about data and time, data quality, and the interpolation method.

Definition: Data Sample

A time-series data sample consists of one or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

For example, suppose that ts.data has the size 5-by-4-by-3 and the time vector has the length 5. Then, the number of samples is 5 and the total number of data values is $5 \times 4 \times 3 = 60$.

Notes About Quality

When Quality is a vector, it must have the same length as the time vector. In this case, each Quality value applies to the corresponding data sample. When Quality is an array, it must have the same size as the data array. In this case, each Quality value applies to the corresponding data value of the ts.data array.

Examples

Example 1 — Using Default Time Vector

Create a timeseries object called 'LaunchData' that contains four data sets, each stored as a column of length 5 and using the default time vector:

```matlab
b = timeseries(rand(5, 4),'Name','LaunchData')
```

Example 2 — Using Uniform Time Vector

Create a timeseries object containing a single data set of length 5 and a time vector starting at 1 and ending at 5:

```matlab
b = timeseries(rand(5,1),[1 2 3 4 5])
```
Example 3

Create a timeseries object called 'FinancialData' containing five data points at a single time point:

```matlab
b = timeseries(rand(1,5),1,'Name','FinancialData')
```

See Also

addsample, tscollection, tsdata.event, tsprops
**Purpose**  
Add title to current axes

**GUI Alternative**  
To create or modify a plot’s title from a GUI, use **Insert Title** from the figure menu. Use the Property Editor, one of the plotting tools, to modify the position, font, and other properties of a legend. For details, see The Property Editor in the MATLAB Graphics documentation.

**Syntax**

- `title('string')`
- `title(fname)`
- `title(...,'PropertyName',PropertyValue,...)`
- `title(axes_handle,...)`
- `h = title(...)`

**Description**  
Each axes graphics object can have one title. The title is located at the top and in the center of the axes.

- `title('string')` outputs the string at the top and in the center of the current axes.
- `title(fname)` evaluates the function that returns a string and displays the string at the top and in the center of the current axes.
- `title(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the text graphics object that `title` creates. Do not use the 'String' text property to set the title string; the content of the title should be given by the first argument.
- `title(axes_handle,...)` adds the title to the specified axes.
- `h = title(...)` returns the handle to the text object used as the title.

**Note**  
The words default, factory, and remove are reserved words that will not appear in a title when quoted as a normal string. In order to display any of these words individually, type '\reserved_word' instead of 'reserved_word'.
**Examples**

Display today’s date in the current axes:

```matlab
title(date)
```

Include a variable’s value in a title:

```matlab
f = 70;
c = (f-32)/1.8;
title(['Temperature is ',num2str(c),'C'])
```

Include a variable’s value in a title and set the color of the title to yellow:

```matlab
n = 3;
title(['Case number #',int2str(n)],'Color','y')
```

Include Greek symbols in a title:

```matlab
title('\text{e}^{\omega\tau} = \cos(\omega\tau) + i\sin(\omega\tau)')
```

Include a superscript character in a title:

```matlab
\text{title}('\alpha^2')
```

Include a subscript character in a title:

```matlab
\text{title}('X_1')
```

The text object `String` property lists the available symbols.

Create a multiline title using a multiline cell array.

```matlab
\text{title}({'First line';'Second line'})
```

**Remarks**

`title` sets the **Title** property of the current axes graphics object to a new text graphics object. See the `text` object **String** property for more information.

**See Also**

`gtext`, `int2str`, `num2str`, `text`, `xlabel`, `ylabel`, `zlabel`
“Annotating Plots” on page 1-94 for related functions

Text Properties for information on setting parameter/value pairs in titles

Adding Titles to Graphs for more information on ways to add titles
Purpose
Convert CDF epoch object to MATLAB datenum

Syntax
n = todatenum(obj)

Description
n = todatenum(obj) converts the CDF epoch object ep_obj into a MATLAB serial date number. Note that a CDF epoch is the number of milliseconds since 01-Jan-0000 whereas a MATLAB datenum is the number of days since 00-Jan-0000.

Examples
Construct a CDF epoch object from a date string, and then convert the object back into a MATLAB date string:

```matlab
dstr = datestr(today)
dstr =
  08-Oct-2003

obj = cdfepoch(dstr)
obj =
    cdfepoch object:
       08-Oct-2003 00:00:00

dstr2 = datestr(todatenum(obj))
dstr2 =
       08-Oct-2003
```

See Also
cdfepoch, cdfinfo, cdfread, cdfwrite, datenum
### Purpose
Toeplitz matrix

### Syntax

T = toeplitz(c,r)

T = toeplitz(r)

### Description
A Toeplitz matrix is defined by one row and one column. A symmetric Toeplitz matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

T = `toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix T having c as its first column and r as its first row. If the first elements of c and r are different, a message is printed and the column element is used.

For a real vector r, T = `toeplitz(r)` returns the symmetric Toeplitz matrix formed from vector r, where r defines the first row of the matrix. For a complex vector r with a real first element, T = `toeplitz(r)` returns the Hermitian Toeplitz matrix formed from r, where r defines the first row of the matrix and r’ defines the first column. When the first element of r is not real, the resulting matrix is Hermitian off the main diagonal, i.e., T<sub>ij</sub> = conj(T<sub>ji</sub>) for i ≠ j.

### Examples
A Toeplitz matrix with diagonal disagreement is

```matlab
c = [1 2 3 4 5];
r = [1.5 2.5 3.5 4.5 5.5];
toeplitz(c,r)
```

Column wins diagonal conflict:
```
ans =
    1.000   2.500   3.500   4.500   5.500
    2.000   1.000   2.500   3.500   4.500
    3.000   2.000   1.000   2.500   3.500
    4.000   3.000   2.000   1.000   2.500
    5.000   4.000   3.000   2.000   1.000
```

### See Also
hankel, kron
**Purpose**

Root directory for specified toolbox

**Syntax**

```
toolboxdir('tbxdirname')
s = toolboxdir('tbxdirname')
s = toolboxdir tbxdirname
```

**Description**

`toolboxdir('tbxdirname')` returns a string that is the absolute path to the specified toolbox, `tbxdirname`, where `tbxdirname` is the directory name for the toolbox.

```
s = toolboxdir('tbxdirname') returns the absolute path to the specified toolbox to the output argument, s.
s = toolboxdir tbxdirname is the command form of the syntax.
```

**Remarks**

`toolboxdir` is particularly useful for MATLAB Compiler software. The base directory of all toolboxes installed with MATLAB software is:

```
matlabroot/toolbox/tbxdirname
```

However, in deployed mode, the base directories of the toolboxes are different. `toolboxdir` returns the correct root directory, whether running from MATLAB or from an application deployed with the MATLAB Compiler software.

**Example**

To obtain the path for the Control System Toolbox software, run

```
s = toolboxdir('control')
```

MATLAB returns

```
s = \myhome\r2009a\matlab\toolbox\control
```

**See Also**

`ctfroot` (in the MATLAB Compiler product), `fullfile`, `matlabroot`, `partialpath`, `path`,

“Managing Files and Working with the Current Directory”
Purpose  Sum of diagonal elements
Syntax  \( b = \text{trace}(A) \)
Description  \( b = \text{trace}(A) \) is the sum of the diagonal elements of the matrix A.
Algorithm  \text{trace} \text{ is a single-statement M-file.}

\[
t = \text{sum} \left( \text{diag}(A) \right);
\]
See Also  \text{det, eig}
transpose (timeseries)

**Purpose**
Transpose timeseries object

**Syntax**

```matlab
ts1 = transpose(ts)
```

**Description**

$ts1 = \text{transpose}(ts)$ returns a new timeseries object $ts1$ with $\text{IsTimeFirst}$ value set to the opposite of what it is for $ts$. For example, if $ts$ has the first data dimension aligned with the time vector, $ts1$ has the last data dimension aligned with the time vector.

**Remarks**

The `transpose` function that is overloaded for the `timeseries` objects does not transpose the data. Instead, this function changes whether the first or the last dimension of the data is aligned with the time vector.

**Note**
To transpose the data, you must transpose the `Data` property of the time series. For example, you can use the syntax `\text{transpose}(ts.\text{Data})` or `(ts.\text{Data}).'`. `Data` must be a 2-D array.

Consider a time series with 10 samples with the property $\text{IsTimeFirst} = \text{True}$. When you transpose this time series, the data size is changed from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

The following table summarizes how the size for time-series data (up to three dimensions) display before and after transposing.

**Data Size Before and After Transposing**

<table>
<thead>
<tr>
<th>Size of Original Data</th>
<th>Size of Transposed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-by-1</td>
<td>1-by-1-by-N</td>
</tr>
<tr>
<td>N-by-M</td>
<td>M-by-1-by-N</td>
</tr>
<tr>
<td>N-by-M-by-L</td>
<td>M-by-L-by-N</td>
</tr>
</tbody>
</table>
**Examples**

Suppose that a timeseries object `ts` has `ts.Data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is set to `true`, which means that the first dimension of the data is aligned with the time vector. `transpose(ts)` modifies the `timeseries` object such that the last dimension of the data is now aligned with the time vector. This permutes the data such that the size of `ts.Data` becomes 3-by-2-by-10.

**See Also**

`ctranspose (timeseries), tsprops`
**Purpose**

Trapezoidal numerical integration

**Syntax**

\[
Z = \text{trapz}(Y) \\
Z = \text{trapz}(X,Y) \\
Z = \text{trapz}(\ldots,\text{dim})
\]

**Description**

- **Z = trapz(Y)** computes an approximation of the integral of \( Y \) via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply \( Z \) by the spacing increment. Input \( Y \) can be complex.

  - If \( Y \) is a vector, \( \text{trapz}(Y) \) is the integral of \( Y \).
  - If \( Y \) is a matrix, \( \text{trapz}(Y) \) is a row vector with the integral over each column.
  - If \( Y \) is a multidimensional array, \( \text{trapz}(Y) \) works across the first nonsingleton dimension.

- **Z = trapz(X,Y)** computes the integral of \( Y \) with respect to \( X \) using trapezoidal integration. Inputs \( X \) and \( Y \) can be complex.

  - If \( X \) is a column vector and \( Y \) an array whose first nonsingleton dimension is \( \text{length}(X) \), \( \text{trapz}(X,Y) \) operates across this dimension.

- **Z = trapz(\ldots,\text{dim})** integrates across the dimension of \( Y \) specified by scalar \( \text{dim} \). The length of \( X \), if given, must be the same as \( \text{size}(Y,\text{dim}) \).

**Examples**

**Example 1**

The exact value of \( \int_{0}^{\pi} \sin(x)dx \) is 2.

To approximate this numerically on a uniformly spaced grid, use

\[
X = 0:pi/100:pi; \\
Y = \sin(X);
\]

Then both

\[
Z = \text{trapz}(X,Y)
\]
and

\[ Z = \frac{\pi}{100} \text{trapz}(Y) \]

produce

\[ Z = 1.9998 \]

**Example 2**

A nonuniformly spaced example is generated by

\[
X = \text{sort}(\text{rand}(1,101) \times \pi);
Y = \sin(X);
Z = \text{trapz}(X,Y);
\]

The result is not as accurate as the uniformly spaced grid. One random sample produced

\[ Z = 1.9984 \]

**Example 3**

This example uses two complex inputs:

\[
z = \exp(1i \times \pi \times (0:100)/100);
\]

\[
\text{trapz}(z, 1./z)
\]

\[
\text{ans} = 0.0000 + 3.1411i
\]

**See Also**

cumsum, cumtrapz
**Purpose**
Lay out tree or forest

**Syntax**

```
[x,y] = treelayout(parent,post)
[x,y,h,s] = treelayout(parent,post)
```

**Description**

`[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with 0 for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

**See Also**
etree, treeplot, etreeplot, symbfact
**Purpose**
Plot picture of tree

**Syntax**
- `treeplot(p)
- `treeplot(p,nodeSpec,edgeSpec)`

**Description**
- `treeplot(p)` plots a picture of a tree given a vector of parent pointers, with `p(i) = 0` for a root.
- `treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `' ' to omit one or both.

**Examples**
To plot a tree with 12 nodes, call `treeplot` with a 12-element input vector. The index of each element in the vector is shown adjacent to each node in the figure below. (These indices are shown only for the point of illustrating the example; they are not part of the `treeplot` output.)

![Tree Plot Example](image)

To generate this plot, set the value of each element in the `nodes` vector to the index of its parent, (setting the parent of the root node to zero).
The node marked 1 in the figure is represented by nodes(1) in the input vector, and because this is the root node which has a parent of zero, you set its value to zero:

\[ \text{nodes}(1) = 0; \quad \% \quad \text{Root node} \]

nodes(2) and nodes(8) are children of nodes(1), so set these elements of the input vector to 1:

\[ \text{nodes}(2) = 1; \quad \text{nodes}(8) = 1; \]

nodes(5:7) are children of nodes(4), so set these elements to 4:

\[ \text{nodes}(5) = 4; \quad \text{nodes}(6) = 4; \quad \text{nodes}(7) = 4; \]

Continue in this manner until each element of the vector identifies its parent. For the plot shown above, the nodes vector now looks like this:

\[ \text{nodes} = [0 \ 1 \ 2 \ 2 \ 4 \ 4 \ 4 \ 1 \ 8 \ 8 \ 10 \ 10]; \]

Now call treeplot to generate the plot:

\[ \text{treeplot}(\text{nodes}) \]

See Also
etree, etreeplot, treelayout
**Purpose**  Lower triangular part of matrix

**Syntax**

\[
\begin{align*}
L &= \text{tril}(X) \\
L &= \text{tril}(X,k)
\end{align*}
\]

**Description**  
\(L = \text{tril}(X)\) returns the lower triangular part of \(X\).
\(L = \text{tril}(X,k)\) returns the elements on and below the \(k\)th diagonal of \(X\). \(k = 0\) is the main diagonal, \(k > 0\) is above the main diagonal, and \(k < 0\) is below the main diagonal.

**Examples**  
\[
\text{tril(ones(4,4),-1)}
\]

\[
\text{ans =}
\begin{bmatrix}
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0
\end{bmatrix}
\]

**See Also**  
\text{diag}, \text{triu}
Purpose
Triangular mesh plot

Syntax
trimesh(Tri,X,Y,Z)
trimesh(Tri,X,Y,Z,C)
trimesh(...'PropertyName',PropertyValue...)
h = trimesh(...)

Description
trimesh(Tri,X,Y,Z) displays triangles defined in the \( m \)-by-3 face
matrix \( \text{T}ri \) as a mesh. Each row of \( \text{T}ri \) defines a single triangular face
by indexing into the vectors or matrices that contain the \( X, Y, \) and \( Z \)
vertices.

\( \text{trimesh(Tri,X,Y,Z,C) \ speci} \)\( \text{ifies color defined by \( C \) in the same
manner as the \text{surf} \) function. The MATLAB software performs a linear
transformation on this data to obtain colors from the current colormap.

\( \text{trimesh(...'PropertyName',PropertyValue...) \ speci} \)\( \text{ifies additional patch property names and values for the patch graphics
object created by the function.}

h = trimesh(...) returns a handle to a patch graphics object.

Example
Create vertex vectors and a face matrix, then create a triangular mesh
plot.

\[
x = \text{rand}(1,50);
y = \text{rand}(1,50);
z = \text{peaks}(6*x-3,6*x-3);
\text{tri} = \text{delaunay}(x,y);
\text{trimesh(tri,x,y,z)}
\]

See Also
patch, tetramesh, triplot, trisurf, delaunay

“Surface and Mesh Creation” on page 1-104 for related functions
Purpose

Numerically evaluate triple integral

Syntax

triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)
triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)

Description

triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax) evaluates the triple integral \(\int_{xmin}^{xmax} \int_{ymin}^{ymax} \int_{zmin}^{zmax} f(x,y,z) \, dz \, dy \, dx\) over the three dimensional rectangular region \(xmin \leq x \leq xmax, ymin \leq y \leq ymax, zmin \leq z \leq zmax\).

fun is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. fun(x,y,z) must accept a vector x and scalars y and z, and return a vector of values of the integrand.

“Parametrizing Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function fun, if necessary.

triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol) uses a tolerance tol instead of the default, which is 1.0e-6.

triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method) uses the quadrature function specified as method, instead of the default quad. Valid values for method are @quadl or the function handle of a user-defined quadrature method that has the same calling sequence as quad and quadl.

Examples

Pass M-file function handle @integrnd to triplequad:

\[
Q = \text{triplequad}(@\text{integrnd},0,\pi,0,1,-1,1);
\]

where the M-file integrnd.m is

```matlab
function f = integrnd(x,y,z)
f = y*sin(x)+z*cos(x);
```

Pass anonymous function handle F to triplequad:

\[
F = @(x,y,z)y*\sin(x)+z*\cos(x);
\]
0 = triplequad(F,0,pi,0,1,-1,1);

This example integrates $y \sin(x) + z \cos(x)$ over the region $0 \leq x \leq \pi, 0 \leq y \leq 1, -1 \leq z \leq 1$. Note that the integrand can be evaluated with a vector $x$ and scalars $y$ and $z$.

See Also
dblquad, quad2d, quad, quadgk, quadl, function handle (@), “Anonymous Functions”
triplot

**Purpose**

2-D triangular plot

**Syntax**

- `triplot(TRI,x,y)`
- `triplot(TRI,x,y,color)`
- `h = triplot(...)`
- `triplot(...,'param','value','param','value'...)`

**Description**

`triplot(TRI,x,y)` displays the triangles defined in the m-by-3 matrix TRI. A row of TRI contains indices into the vectors x and y that define a single triangle. The default line color is blue.

`triplot(TRI,x,y,color)` uses the string `color` as the line color. `color` can also be a line specification. See `ColorSpec` for a list of valid color strings. See `LineSpec` for information about line specifications.

`h = triplot(...)` returns a vector of handles to the displayed triangles.

`triplot(...,'param','value','param','value'...)` allows additional line property name/property value pairs to be used when creating the plot. See `Line Properties` for information about the available properties.

**Examples**

This code plots the Delaunay triangulation for 10 randomly generated points.

```matlab
rand('state',7);
x = rand(1,10);
y = rand(1,10);
TRI = delaunay(x,y);
triplot(TRI,x,y,'red')
```
See Also

ColorSpec, delaunay, line, Line Properties, LineSpec, plot, trimesh, trisurf
**TriRep class**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Triangulation representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>TriRep provides topological and geometric queries for triangulations in 2-D and 3-D space. For example, for triangular meshes you can query triangles attached to a vertex, triangles that share an edge, neighbor information, circumcenters, or other features. You can create a TriRep directly using existing triangulation data. Alternatively, you can create a Delaunay triangulation, via DelaunayTri, which provides access to the TriRep functionality.</td>
</tr>
<tr>
<td><strong>Construction</strong></td>
<td>TriRep</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td>baryToCart</td>
</tr>
<tr>
<td></td>
<td>cartToBary</td>
</tr>
<tr>
<td></td>
<td>circumcenters</td>
</tr>
<tr>
<td></td>
<td>edgeAttachments</td>
</tr>
<tr>
<td></td>
<td>edges</td>
</tr>
<tr>
<td></td>
<td>faceNormals</td>
</tr>
<tr>
<td></td>
<td>featureEdges</td>
</tr>
<tr>
<td></td>
<td>freeBoundary</td>
</tr>
<tr>
<td></td>
<td>incenters</td>
</tr>
<tr>
<td></td>
<td>isEdge</td>
</tr>
</tbody>
</table>
TriRep class

neighbors
Simplex neighbor information
size
Size of triangulation matrix
vertexAttachments
Return simplices attached to specified vertices

Properties

<table>
<thead>
<tr>
<th>X</th>
<th>Coordinates of the points in the triangulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangulation</td>
<td>Triangulation data structure</td>
</tr>
</tbody>
</table>

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Indexing

TriRep objects support indexing into the triangulation using parentheses (). The syntax is the same as for arrays.

Examples

Load a 2-D triangulation and use the TriRep constructor to build an array of the free boundary edges:

```matlab
load trimesh2d

trep = TriRep(tri, x,y);
fe = freeBoundary(trep)';
triplot(trep);
```

This loads triangulation tri and vertex coordinates x, y:
You can add the free edges \texttt{fe} to the plot:

```matlab
hold on;
plot(x(fe), y(fe), 'r', 'LineWidth', 2);
hold off;
axis([-50 350 -50 350]);
axis equal;
```
See Also

“Triangulation Representations” — How to use triangulation representations
DelaunayTri class
TriScatteredInterp class
TriRep

**Purpose**
Triangulation representation

**Syntax**

```
TR = TriRep(TRI, X, Y)
TR = TriRep(TRI, X, Y, Z)
TR = TriRep(TRI, X)
```

**Description**

TR = TriRep(TRI, X, Y) creates a 2-D triangulation representation from the triangulation matrix TRI and the vertex coordinates (X, Y). TRI is an m-by-3 matrix that defines the triangulation in face-vertex format, where m is the number of triangles. Each row of TRI is a triangle defined by indices into the column vector of vertex coordinates (X, Y).

TR = TriRep(TRI, X, Y, Z) creates a 3-D triangulation representation from the triangulation matrix TRI and the vertex coordinates (X, Y, Z). TRI is an m-by-3 or m-by-4 matrix that defines the triangulation in simplex-vertex format, where where m is the number of simplices; triangles or tetrahedra in this case. Each row of TRI is a simplex defined by indices into the column vector of vertex coordinates (X, Y, Z).

TR = TriRep(TRI, X) creates a triangulation representation from the triangulation matrix TRI and the vertex coordinates X. TRI is an m-by-n matrix that defines the triangulation in simplex-vertex format, where m is the number of simplices and n is the number of vertices per simplex. Each row of TRI is a simplex defined by indices into the array of vertex coordinates X. X is an mpts-by-ndim matrix where mpts is the number of points and ndim is the dimension of the space where the points reside, where 2 ≤ ndim ≤ 3.

**Examples**

Load a 3-D tetrahedral triangulation compute the free boundary. First, load triangulation tet and vertex coordinates X.

```
load tetmesh
```

Create the triangulation representation and compute the free boundary.

```
trep = TriRep(tet, X);
[tri, Xb] = freeBoundary(trep);
```
See Also

TriScatteredInterp
“Computational Geometry”— A guide to MATLAB’s object-oriented and functional capabilities for computational geometry.
TriScatteredInterp class

Purpose
Interpolate scattered data

Description
A scattered data set defined by locations X and corresponding values V can be interpolated using a Delaunay triangulation of X. This produces a surface of the form \( V = F(X) \). The surface can be evaluated at any query location \( Q_X \), using \( Q_V = F(Q_X) \), where \( Q_X \) lies within the convex hull of \( X \). The interpolant \( F \) always goes through the data points specified by the sample.

Definitions
The Delaunay triangulation of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set. The convex hull of a set of points is the smallest convex set containing all points of the original set. These definitions extend naturally to higher dimensions.

Construction
TriScatteredInterp

Properties

<table>
<thead>
<tr>
<th>X</th>
<th>Defines locations of scattered data points in 2-D or 3-D space.</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Defines value associated with each data point.</td>
</tr>
<tr>
<td>Method</td>
<td>Defines method used to interpolate the data.</td>
</tr>
<tr>
<td></td>
<td>natural</td>
</tr>
<tr>
<td></td>
<td>linear</td>
</tr>
<tr>
<td></td>
<td>nearest</td>
</tr>
</tbody>
</table>

Copy Semantics
Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.
Examples

Create a data set:

\[
\begin{align*}
  x &= \text{rand}(100,1)*4-2; \\
  y &= \text{rand}(100,1)*4-2; \\
  z &= x.*\exp(-x.^2-y.^2);
\end{align*}
\]

Construct the interpolant:

\[
F = \text{TriScatteredInterp}(x,y,z);
\]

Evaluate the interpolant at the locations (qx, qy). The corresponding value at these locations is qz:

\[
\begin{align*}
  \text{ti} &= -2:.25:2; \\
  [\text{qx},\text{qy}] &= \text{meshgrid}(\text{ti},\text{ti}); \\
  \text{qz} &= F(\text{qx},\text{qy}); \\
  \text{mesh}(\text{qx},\text{qy},\text{qz}); \\
  \text{hold} \text{ on}; \\
  \text{plot3}(x,y,z,'o');
\end{align*}
\]
**TriScatteredInterp class**

**See Also**

“Triangulation-Based Interpolation”—Object-oriented syntax for interpolation.
“Triangulations and Scattered Data”—Functional syntax for interpolation.
DelaunayTri
interp1
interp2
interp3
meshgrid
Syntax

\[
F = \text{TriScatteredInterp}(\cdot)
\]

\[
F = \text{TriScatteredInterp}(X, V)
\]

\[
F = \text{TriScatteredInterp}(X, Y, V)
\]

\[
F = \text{TriScatteredInterp}(X, Y, Z, V)
\]

\[
F = \text{TriScatteredInterp}(\text{DT}, V)
\]

\[
F = \text{TriScatteredInterp}(\ldots, \text{method})
\]

Description

\(F = \text{TriScatteredInterp}(\cdot)\) creates an empty scattered data interpolant. This can subsequently be initialized with sample data points and values \((X_{\text{data}}, V_{\text{data}})\) via \(F.X = X_{\text{data}}\) and \(F.V = V_{\text{data}}\).

\(F = \text{TriScatteredInterp}(X, V)\) creates an interpolant that fits a surface of the form \(V = F(X)\) to the scattered data in \((X, V)\). \(X\) is a matrix of size \(m_{\text{pts}}\)-by-\(n_{\text{dim}}\), where \(m_{\text{pts}}\) is the number of points and \(n_{\text{dim}}\) is the dimension of the space where the points reside, \(n_{\text{dim}} \geq 2\). The column vector \(V\) defines the values at \(X\), where the length of \(V\) equals \(m_{\text{pts}}\).

\(F = \text{TriScatteredInterp}(X, Y, V)\) and \(F = \text{TriScatteredInterp}(X, Y, Z, V)\) allow the data point locations to be specified in alternative column vector format when working in 2-D and 3-D.

\(F = \text{TriScatteredInterp}(\text{DT}, V)\) uses the specified DelaunayTri object \(\text{DT}\) as a basis for computing the interpolant. The matrix \(\text{DT}.X\) is of size \(m_{\text{pts}}\)-by-\(n_{\text{dim}}\), where \(m_{\text{pts}}\) is the number of points and \(n_{\text{dim}}\) is the dimension of the space where the points reside, \(2 \leq n_{\text{dim}} \leq 3\). \(V\) is a column vector that defines the values at \(\text{DT}.X\), where the length of \(V\) equals \(m_{\text{pts}}\).

\(F = \text{TriScatteredInterp}(\ldots, \text{method})\) allows selection of the technique \(\text{method}\) used to interpolate the data.

Inputs

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X)</td>
<td>Matrix of size (m_{\text{pts}})-by-(n_{\text{dim}}), where (m_{\text{pts}}) is the number of points and (n_{\text{dim}}) is the dimension of the space where the points reside.</td>
</tr>
<tr>
<td>(V)</td>
<td>Column vector that defines the values at (X), where the length of (V) equals (m_{\text{pts}}).</td>
</tr>
</tbody>
</table>
**TriScatteredInterp**

<table>
<thead>
<tr>
<th>DT</th>
<th>Delaunay triangulation of the scattered data locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>natural</td>
</tr>
<tr>
<td></td>
<td>linear</td>
</tr>
<tr>
<td></td>
<td>nearest</td>
</tr>
</tbody>
</table>

**Outputs**

| F | Creates an interpolant that fits a surface of the form $V = F(X)$ to the scattered data. |

**Evaluation**

To evaluate the interpolant, express the statement in Monge’s form $V = F(x)$, $V = F(x, y)$, or $V = F(x, y, z)$.

**Definitions**

The *Delaunay triangulation* of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set.

**Examples**

Create a data set:

```matlab
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```matlab
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations $(qx, qy)$. The corresponding value at these locations is $qz$.

```matlab
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
```
plot3(x,y,z,'o');

See Also

DelaunayTri
interp1
interp2
interp3
meshgrid
trisurf

**Purpose**
Triangular surface plot

**Syntax**
```
trisurf(Tri,X,Y,Z)
trisurf(Tri,X,Y,Z,C)
trisurf(...'PropertyName',PropertyValue...)
```

**Description**
`trisurf(Tri,X,Y,Z)` displays triangles defined in the `m`-by-3 face matrix `Tri` as a surface. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices.

`trisurf(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. The MATLAB software performs a linear transformation on this data to obtain colors from the current colormap.

`trisurf(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trisurf(...)` returns a patch handle.

**Example**
Create vertex vectors and a face matrix, then create a triangular surface plot.
```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x-3,6*x-3);
tri = delaunay(x,y);
trisurf(tri,x,y,z)
```

**See Also**
patch, surf, tetramesh, trimesh, triplot, delaunay

“Surface and Mesh Creation” on page 1-104 for related functions
Purpose
Upper triangular part of matrix

Syntax
U = triu(X)
U = triu(X,k)

Description
U = triu(X) returns the upper triangular part of X.
U = triu(X,k) returns the element on and above the kth diagonal of X.
k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

Examples
triu(ones(4,4),-1)

ans =

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

See Also
diag, tril
true

**Purpose**

Logical 1 (true)

**Syntax**

```
true
true(n)
true(m, n)
true(m, n, p, ...)
true(size(A))
```

**Description**

true is shorthand for logical 1.

true(n) is an n-by-n matrix of logical ones.

true(m, n) or true([m, n]) is an m-by-n matrix of logical ones.

true(m, n, p, ...) or true([m n p ...]) is an m-by-n-by-p-by-... array of logical ones.

**Note**
The size inputs m, n, p, ... should be nonnegative integers. Negative integers are treated as 0.

true(size(A)) is an array of logical ones that is the same size as array A.

**Remarks**

true(n) is much faster and more memory efficient than logical(ones(n)).

**See Also**

false, logical
**Purpose**

Attempt to execute block of code, and catch errors

**Syntax**

```
try
```

**Description**

`try` marks the start of a *try block* in a *try-catch* statement. If the MATLAB software detects an error while executing code in the try block, it immediately jumps to the start of the respective *catch block* and executes the error handling code in that block.

A *try-catch* statement is a programming device that enables you to define how certain errors are to be handled in your program. This bypasses the default MATLAB error-handling mechanism when these errors are detected. The *try-catch* statement consists of two blocks of MATLAB code, a *try block* and a *catch block*, delimited by the keywords `try`, `catch`, and `end`:

```
try
    MATLAB commands % Try block
catch ME
    MATLAB commands % Catch block
end
```

Each of these blocks consists of one or more MATLAB commands. The *try block* is just another piece of your program code; the commands in this block execute just like any other part of your program. Any errors MATLAB encounters in the *try block* are dealt with by the respective *catch block*. This is where you write your error-handling code. If the *try block* executes without error, MATLAB skips the *catch block* entirely. If an error occurs while executing the *catch block*, the program terminates unless this error is caught by another *try-catch* block.

Specifying the `try`, `catch`, and `end` commands, as well as the commands that make up 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 =
```
where the term `matlabroot` represents the string returned by the `matlabroot` function.

**Examples**

The catch block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension. This is done using a `try-catch` statement that is nested within the original `try-catch`.

```matlab
function d_in = read_image(filename)
[path name ext] = fileparts(filename);
try
    fid = fopen(filename, 'r');
    d_in = fread(fid);
catch ME1
    % Get last segment of the error message identifier.
    idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

    % Did the read fail because the file could not be found?
    if strcmp(idSegLast, 'InvalidFid') && ~exist(filename, 'file')
        % Yes. Try modifying the filename extension.
        switch ext
            case '.jpg' % Change jpg to jpeg
                filename = strrep(filename, '.jpg', '.jpeg')
            case '.jpeg' % Change jpeg to jpg
                filename = strrep(filename, '.jpeg', '.jpg')
            case '.tif' % Change tif to tiff
                filename = strrep(filename, '.tif', '.tiff')
            case '.tiff' % Change tiff to tif
```
filename = strrep(filename, '.tiff', '.tif')
otherwise
    fprintf('File %s not found\n', filename);
    rethrow(ME1);
end

% Try again, with modified filenames.
try
    fid = fopen(filename, 'r');
    d_in = fread(fid);
    catch ME2
        fprintf('Unable to access file %s\n', filename);
        ME2 = addCause(ME2, ME1);
        rethrow(ME2)
    end
end
end

See Also

catch, rethrow, end, lasterror, eval, evalin
**Purpose**
Create `tscollection` object

**Syntax**
- `tsc = tscollection(TimeSeries)`
- `tsc = tscollection(Time)`
- `tsc = tscollection(Time,TimeSeries,'Parameter',Value,...)`

**Description**
- `tsc = tscollection(TimeSeries)` creates a `tscollection` object `tsc` with one or more `timeseries` objects already in the MATLAB workspace. The argument `TimeSeries` can be a
  - Single `timeseries` object
  - Cell array of `timeseries` objects

- `tsc = tscollection(Time)` creates an empty `tscollection` object with the time vector `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

- `tsc = tscollection(Time,TimeSeries,'Parameter',Value,...)` creates a `tscollection` object with optional parameter-value pairs you enter after the `Time` and `TimeSeries` arguments. You can specify the following parameter:
  - Name — String that specifies the name of this `tscollection` object

**Remarks**
**Definition: Time Series Collection**
A time series collection object is a MATLAB variable that groups several time series with a common time vector. The time series that you include in the collection are called members of this collection.

**Properties of Time Series Collection Objects**
This table lists the properties of the `tscollection` object. You can specify the `Time`, `TimeSeries`, and `Name` properties as input arguments in the constructor.
### tscollection

**Property** | **Description**  
---|---  
Name | tscollection name as a string. This can differ from the tscollection name in the MATLAB workspace.  
Time | When TimeInfo.StartDate is empty, values are measured relative to 0. When TimeInfo.StartDate is defined, values represent date strings measured relative to the StartDate.  
| The length of Time must be the same as the first or the last dimension of Data for each collection.  
TimeInfo | Contains fields for contextual information about Time:  
| • **Units** — Time units with any of the following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', 'nanoseconds'  
| • **Start** — Start time  
| • **End** — End time (read only)  
| • **Increment** — Interval between subsequent time values. NaN when times are not uniformly sampled.  
| • **Length** — Length of the time vector (read only)  
| • **Format** — String defining the date string display format. See datestr.  
| • **StartDate** — Date string defining the reference date.  
| See setabstime (tscollection).  
| • **UserData** — Any additional user-defined information

### Examples

The following example shows how to create a tscollection object.

1 Import the sample data.

   ```matlab
   load count.dat
   ```
2 Create three `timeseries` objects to store each set of data:

```matlab
count1 = timeseries(count(:,1),1:24,'name', 'ts1');
count2 = timeseries(count(:,2),1:24,'name', 'ts2');
```

3 Create a `tscollection` object named `tsc` and add to it two out of three time series already in the MATLAB workspace, by using the following syntax:

```matlab
tsc = tscollection({count1 count2},'name','tsc')
```

**See Also**
`addts`, `datestr`, `setabstime` (tscollection), `timeseries`, `tsprops`
Purpose

Construct event object for timeseries object

Syntax

e = tsdata.event(Name,Time)
e = tsdata.event(Name,Time,'Datenum')

Description

e = tsdata.event(Name,Time) creates an event object with the specified Name that occurs at the time Time. Time can either be a real value or a date string.

e = tsdata.event(Name,Time,'Datenum') uses 'Datenum' to indicate that the Time value is a serial date number generated by the datenum function. The Time value is converted to a date string after the event is created.

Remarks

You add events by using the addevent method.

Fields of the tsdata.event object include the following:

- **EventData** — MATLAB array that stores any user-defined information about the event
- **Name** — String that specifies the name of the event
- **Time** — Time value when this event occurs, specified as a real number
- **Units** — Time units
- **StartDate** — A reference date, specified in MATLAB datetstr format. StartDate is empty when you have a numerical (non-date-string) time vector.
**Purpose**
Search for enclosing Delaunay triangle

**Syntax**
\[ T = \text{tsearch}(x,y,\text{TRI},x_i,y_i) \]

**Description**
\[ T = \text{tsearch}(x,y,\text{TRI},x_i,y_i) \] returns an index into the rows of \( \text{TRI} \) for each point in \( x_i, y_i \). The \text{tsearch} command returns \text{NaN} for all points outside the convex hull. Requires a triangulation \( \text{TRI} \) of the points \( x,y \) obtained from \text{delaunay}.

**See Also**
delaunay, delaunayn, dsearch, tsearchn
**Purpose**
N-D closest simplex search

**Syntax**
\[
t = \text{tsearchn}(X, \text{TES}, XI)
\]
\[
[t, P] = \text{tsearchn}(X, \text{TES}, XI)
\]

**Description**
\( t = \text{tsearchn}(X, \text{TES}, XI) \) returns the indices \( t \) of the enclosing simplex of the Delaunay tessellation \( \text{TES} \) for each point in \( XI \). \( X \) is an \( m \)-by-\( n \) matrix, representing \( m \) points in \( N \)-dimensional space. \( XI \) is a \( p \)-by-\( n \) matrix, representing \( p \) points in \( N \)-dimensional space. \( \text{tsearchn} \) returns \( \text{NaN} \) for all points outside the convex hull of \( X \). \( \text{tsearchn} \) requires a tessellation \( \text{TES} \) of the points \( X \) obtained from \text{delaunayn}.

\([t, P] = \text{tsearchn}(X, \text{TES}, XI)\) also returns the barycentric coordinate \( P \) of \( XI \) in the simplex \( \text{TES} \). \( P \) is a \( p \)-by-\( n+1 \) matrix. Each row of \( P \) is the Barycentric coordinate of the corresponding point in \( XI \). It is useful for interpolation.

**Algorithm**
\( \text{tsearchn} \) is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

**See Also**
delaunayn, griddatan, tsearch

**Reference**
**tsprops**

**Purpose**
Help on timeseries object properties

**Syntax**
help timeseries/tsprops

**Description**
help timeseries/tsprops lists the properties of the timeseries object and briefly describes each property.

**Time Series Object Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Time-series data, where each data sample corresponds to a specific time. The data can be a scalar, a vector, or a multidimensional array. Either the first or last dimension of the data must be aligned with Time. By default, NaNs are used to represent missing or unspecified data. Set the TreatNaNasMissing property to determine how missing data is treated in calculations.</td>
</tr>
<tr>
<td>DataInfo</td>
<td>Contains fields for storing contextual information about Data:</td>
</tr>
<tr>
<td></td>
<td>• Unit — String that specifies data units</td>
</tr>
<tr>
<td></td>
<td>• Interpolation — A tsdata.interpolation object that specifies the interpolation method for this time series. Fields of the tsdata.interpolation object include:</td>
</tr>
<tr>
<td></td>
<td>• Fhandle — Function handle to a user-defined interpolation function</td>
</tr>
<tr>
<td></td>
<td>• Name — String that specifies the name of the interpolation method. Predefined methods include 'linear' and 'zoh' (zero-order hold). 'linear' is the default.</td>
</tr>
<tr>
<td></td>
<td>• UserData — Any user-defined information entered as a string</td>
</tr>
</tbody>
</table>
## Time Series Object Properties (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>An array of <code>tsdata.event</code> objects that stores event information for this time series. You add events by using the <code>addevent</code> method. Fields of the <code>tsdata.event</code> object include the following:</td>
</tr>
<tr>
<td></td>
<td>- <strong>EventData</strong> — Any user-defined information about the event</td>
</tr>
<tr>
<td></td>
<td>- <strong>Name</strong> — String that specifies the name of the event</td>
</tr>
<tr>
<td></td>
<td>- <strong>Time</strong> — Time value when this event occurs, specified as a real number or a date string</td>
</tr>
<tr>
<td></td>
<td>- <strong>Units</strong> — Time units</td>
</tr>
<tr>
<td></td>
<td>- <strong>StartDate</strong> — A reference date specified in MATLAB date-string format. <code>StartDate</code> is empty when you have a numerical (non-date-string) time vector.</td>
</tr>
</tbody>
</table>
Time Series Object Properties (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsTimeFirst</td>
<td>Logical value (true or false) specifies whether the first or last dimension of the Data array is aligned with the time vector.</td>
</tr>
<tr>
<td></td>
<td>You can set this property when the Data array is square and it is ambiguous which dimension is aligned with time. By default, the first Data dimension that matches the length of the time vector is aligned with the time vector.</td>
</tr>
<tr>
<td></td>
<td>When you set this property to:</td>
</tr>
<tr>
<td></td>
<td>• true — The first dimension of the data array is aligned with the time vector. For example:</td>
</tr>
<tr>
<td></td>
<td>ts=timeseries(rand(3,3),1:3,'IsTimeFirst',true);</td>
</tr>
<tr>
<td></td>
<td>• false — The last dimension of the data array is aligned with the time vector. For example:</td>
</tr>
<tr>
<td></td>
<td>ts=timeseries(rand(3,3),1:3,'IsTimeFirst',false);</td>
</tr>
<tr>
<td></td>
<td>After a time series is created, this property is read only.</td>
</tr>
<tr>
<td>Name</td>
<td>Time-series name entered as a string. This name can differ from the name of the time-series variable in the MATLAB workspace.</td>
</tr>
<tr>
<td>Quality</td>
<td>An integer vector or array containing values -128 to 127 that specifies the quality in terms of codes defined by QualityInfo.Code.</td>
</tr>
<tr>
<td></td>
<td>When Quality is a vector, it must have the same length as the time vector. In this case, each Quality value applies to a corresponding data sample.</td>
</tr>
<tr>
<td></td>
<td>When Quality is an array, it must have the same size as the data array. In this case, each Quality value applies to the corresponding value of the data array.</td>
</tr>
</tbody>
</table>
### Time Series Object Properties (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QualityInfo</td>
<td>Provides a lookup table that converts numerical Quality codes to readable descriptions. QualityInfo fields include the following:</td>
</tr>
<tr>
<td></td>
<td>• Code — Integer vector containing values -128 to 127 that define the “dictionary” of quality codes, which you can assign to each Data value by using the Quality property</td>
</tr>
<tr>
<td></td>
<td>• Description — Cell vector of strings, where each element provides a readable description of the associated quality Code</td>
</tr>
<tr>
<td></td>
<td>• UserData — Stores any additional user-defined information</td>
</tr>
<tr>
<td></td>
<td>Lengths of Code and Description must match.</td>
</tr>
<tr>
<td>Time</td>
<td>Array of time values.</td>
</tr>
<tr>
<td></td>
<td>When TimeInfo.StartDate is empty, the numerical Time values are measured relative to 0 in specified units. When TimeInfo.StartDate is defined, the time values are date strings measured relative to the StartDate in specified units.</td>
</tr>
<tr>
<td></td>
<td>The length of Time must be the same as either the first or the last dimension of Data.</td>
</tr>
</tbody>
</table>
Time Series Object Properties (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeInfo</td>
<td>Uses the following fields for storing contextual information about Time:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Units</strong> — Time units can have any of following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', or 'nanoseconds'</td>
</tr>
<tr>
<td></td>
<td>• <strong>Start</strong> — Start time</td>
</tr>
<tr>
<td></td>
<td>• <strong>End</strong> — End time (read only)</td>
</tr>
<tr>
<td></td>
<td>• <strong>Increment</strong> — Interval between two subsequent time values</td>
</tr>
<tr>
<td></td>
<td>• <strong>Length</strong> — Length of the time vector (read only)</td>
</tr>
<tr>
<td></td>
<td>• <strong>Format</strong> — String defining the date string display format. See the MATLAB <code>datestr</code> function reference page for more information.</td>
</tr>
<tr>
<td></td>
<td>• <strong>StartDate</strong> — Date string defining the reference date. See the MATLAB <code>setabstime</code> function reference page for more information.</td>
</tr>
<tr>
<td></td>
<td>• <strong>UserData</strong> — Stores any additional user-defined information</td>
</tr>
<tr>
<td>TreatNaNasMissing</td>
<td>Logical value that specifies how to treat NaN values in <strong>Data</strong>:</td>
</tr>
<tr>
<td></td>
<td>• <strong>true</strong> — (Default) Treat all NaN values as missing data except during statistical calculations.</td>
</tr>
<tr>
<td></td>
<td>• <strong>false</strong> — Include NaN values in statistical calculations, in which case NaN values are propagated to the result.</td>
</tr>
</tbody>
</table>
See Also

datestr, get (timeseries), set (timeseries), setabstime (timeseries)
### Purpose

Open Time Series Tools GUI

### Syntax

```
tstool
stool(ts)
tstool(tsc)
tstool(sldata)
tstool(ModelDataLogs,'replace')
```

### Description

**tstool** starts the Time Series Tools GUI without loading any data.

**tstool**(ts) starts the Time Series Tools GUI and loads the time-series object ts from the MATLAB workspace.

**tstool**(tsc) starts the Time Series Tools GUI and loads the time-series collection object tsc from the MATLAB workspace.

**tstool**(sldata) starts the Time Series Tools GUI and loads the logged-signal data sldata from a Simulink model. If a Simulink logged signal Name property contains a /, the entire logged signal, including all levels of the signal hierarchy, is not imported into Time Series Tools.

**tstool**(ModelDataLogs,'replace') replaces the logged-signal data object ModelDataLogs in the Time Series Tools GUI with an updated logged signal after you rerun the Simulink model. Use this command to update the ModelDataLogs object in the Time Series Tools GUI if you change the model or the logged-signal data settings.

### See Also

timeseries, tscollection
**Purpose**
Display contents of file

**Syntax**
type('filename')
type filename

**Description**
type('filename') displays the contents of the specified file in the MATLAB Command Window. Use the full path for filename, or use a MATLAB relative partialpath.

If you do not specify a file extension and there is no filename file without an extension, the type function adds the .m extension by default. The type function checks the directories specified in the MATLAB search path, which makes it convenient for listing the contents of M-files on the screen. Use type with more on to see the listing one screen at a time.

type filename is the command form of the syntax.

**Examples**
type('foo.bar') lists the contents of the file foo.bar.
type foo lists the contents of the file foo. If foo does not exist, type foo lists the contents of the file foo.m.

**See Also**
cd, dbtype, delete, dir, more, partialpath, path, what, who
typecast

**Purpose**
Convert data types without changing underlying data

**Syntax**
Y = typecast(X, type)

**Description**
Y = typecast(X, type) converts a numeric value in X to the data type specified by type. Input X must be a full, noncomplex, numeric scalar or vector. The type input is a string set to one of the following: 'uint8', 'int8', 'uint16', 'int16', 'uint32', 'int32', 'uint64', 'int64', 'single', or 'double'.

typecast is different from the MATLAB cast function in that it does not alter the input data. typecast always returns the same number of bytes in the output Y as were in the input X. For example, casting the 16-bit integer 1000 to uint8 with typecast returns the full 16 bits in two 8-bit segments (3 and 232) thus keeping its original value (3*256 + 232 = 1000). The cast function, on the other hand, truncates the input value to 255.

The output of typecast can be formatted differently depending on what system you use it on. Some computer systems store data starting with its most significant byte (an ordering called big-endian), while others start with the least significant byte (called little-endian).

**Note**
MATLAB issues an error if X contains fewer values than are needed to make an output value.

**Examples**

**Example 1**
This example converts between data types of the same size:

```matlab
  typecast(uint8(255), 'int8')
  ans =
       -1

  typecast(int16(-1), 'uint16')
  ans =
```
**Example 2**

Set \( X \) to a 1-by-3 vector of 32-bit integers, then cast it to an 8-bit integer type:

\[
X = \text{uint32}([1, 255, 256])
\]

\[
X = \\
1 \\
255 \\
256
\]

Running this on a little-endian system produces the following results. Each 32-bit value is divided up into four 8-bit segments:

\[
Y = \text{typecast}(X, 'uint8')
\]

\[
Y = \\
1 \\
0 \\
0 \\
255 \\
0 \\
0 \\
0 \\
0 \\
1 \\
0 \\
0
\]

The third element of \( X \), 256, exceeds the 8 bits that it is being converted to in \( Y(9) \) and thus overflows to \( Y(10) \):

\[
Y(9:12)
\]

\[
\text{ans} = \\
0 \\
1 \\
0 \\
0
\]

Note that \( \text{length}(Y) \) is equal to \( 4 \times \text{length}(X) \). Also note the difference between the output of typecast versus that of cast:

\[
Z = \text{cast}(X, 'uint8')
\]

\[
Z = \\
1 \\
255 \\
255
\]

**Example 3**

This example casts a smaller data type (\texttt{uint8}) into a larger one (\texttt{uint16}). Displaying the numbers in hexadecimal format makes it easier to see just how the data is being rearranged:

\[
\text{format hex}
\]

\[
X = \text{uint8}([44, 55, 66, 77])
\]

\[
X = \\
44 \\
55 \\
66 \\
77
\]
The first `typecast` is done on a big-endian system. The four 8-bit segments of the input data are combined to produce two 16-bit segments:

```matlab
Y = typecast(X, 'uint16')
Y =
  2c37 424d
```

The second is done on a little-endian system. Note the difference in byte ordering:

```matlab
Y = typecast(X, 'uint16')
Y =
  372c 4d42
```

You can format the little-endian output into big-endian (and vice versa) using the `swapbytes` function:

```matlab
Y = swapbytes(typecast(X, 'uint16'))
Y =
  2c37 424d
```

**Example 4**

This example attempts to make a 32-bit value from a vector of three 8-bit values. MATLAB issues an error because there are an insufficient number of bytes in the input:

```matlab
format hex

typecast(uint8([120 86 52]), 'uint32')
??? Too few input values to make output type.
```

```
Error in ==> typecast at 29
out = typecastc(in, datatype);
```

Repeat the example, but with a vector of four 8-bit values, and it returns the expected answer:
typecast(uint8([120 86 52 18]), 'uint32')
ans =
   12345678

See Also: cast, class, swapbytes
uibuttongroup

**Purpose**
Create container object to exclusively manage radio buttons and toggle buttons

**Syntax**
```
uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,...)
handle = uibuttongroup(...)```

**Description**
A uibuttongroup groups components and manages exclusive selection behavior for radio buttons and toggle buttons that it contains. It can also contain other user interface controls, axes, uipanels, and uibuttongroups. It cannot contain ActiveX controls.

```
uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,...)
```
creates a visible container component in the current figure window. This component manages exclusive selection behavior for uicontrols of style radiobutton and togglebutton.

```
handle = uibuttongroup(...) creates a uibuttongroup object and returns a handle to it in handle.
```

A uibuttongroup object can have axes, uicontrol, uipanel, and uibuttongroup objects as children. However, only uicontrols of style radiobutton and togglebutton are managed by the component.

When programming a button group, you do not code callbacks for the individual buttons; instead, use its SelectionChangeFcn callback to manage responses to selections. The following example illustrates how you use uibuttongroup event data to do this.

For the children of a uibuttongroup object, the Position property is interpreted relative to the button group. If you move the button group, the children automatically move with it and maintain their positions in the button group.

If you have a button group that contains a set of radio buttons and toggle buttons and you want:

- An immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group’s SelectionChangeFcn callback function,
not in the individual toggle button Callback functions. See the SelectionChangeFcn property and the example on this reference page for more information.

- Another component such as a push button to base its action on the selection, then that component’s Callback callback can get the handle of the selected radio button or toggle button from the button group’s SelectedObject property.

Use the Parent property to specify the parent as a figure, uipanel, or uibuttongroup. If you do not specify a parent, uibuttongroup adds the button group to the current figure. If no figure exists, one is created.

See the Uibuttongroup Properties reference page for more information.

After creating a uibuttongroup, you can set and query its property values using set and get. Run get(handle) to see a list of properties and their current values. Run set(handle) to see a list of object properties you can set and their legal values.

**Remarks**

If you set the Visible property of a uibuttongroup object to ‘off’, any child objects it contains (buttons, button groups, etc.) become invisible along with the uibuttongroup panel itself. However, doing this does not affect the settings of the Visible property of any of its child objects, even though all of them remain invisible until the button group’s visibility is set to ‘on’. uipanel components also behave in this manner.

**Examples**

This example creates a uibuttongroup with three radiobuttons. It manages the radiobuttons with the SelectionChangeFcn callback, selcbk.

When you select a new radio button, selcbk displays the uibuttongroup handle on one line, the EventName, OldValue, and NewValue fields of the event data structure on a second line, and the value of the SelectedObject property on a third line.

```matlab
% Create the button group.
h = uibuttongroup('visible','off','Position',[0 0 .2 1]);
```
% Create three radio buttons in the button group.
u0 = uicontrol('Style','Radio','String','Option 1',... 'pos',[10 350 100 30],'parent',h,'HandleVisibility','off');
u1 = uicontrol('Style','Radio','String','Option 2',... 'pos',[10 250 100 30],'parent',h,'HandleVisibility','off');
u2 = uicontrol('Style','Radio','String','Option 3',... 'pos',[10 150 100 30],'parent',h,'HandleVisibility','off');
% Initialize some button group properties.
set(h,'SelectionChangeFcn',@selcbk);
set(h,'SelectedObject',[]); % No selection
set(h,'Visible','on');

For the SelectionChangeFcn callback, selcbk, the source and event data structure arguments are available only if selcbk is called using a function handle. See SelectionChangeFcn for more information.

function selcbk(source,eventdata)
disp(source);
disp([eventdata.EventName,' ',... get(eventdata.OldValue,'String'),' ',... get(eventdata.NewValue,'String')]);
disp(get(get(source,'SelectedObject'),'String'));

2-3810
If you click Option 2 with no option selected, the `SelectionChangeFcn` callback, `selcbk`, displays:

```
3.0011
SelectionChanged Option 2
Option 2
```

If you then click Option 1, the `SelectionChangeFcn` callback, `selcbk`, displays:

```
3.0011
SelectionChanged Option 2 Option 1
Option 1
```
uibuttongroup

See Also  uicontrol, uipanel
Uibuttongroup Properties

Purpose

Describe button group properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the inspect function at the command line.

- The set and get functions enable you to set and query the values of properties.

Uibuttongroup takes its default property values from uipanel. To set a uibuttongroup default property value, set the default for the corresponding uipanel property. Note that you can set no default values for the uibuttongroup SelectedObject and SelectionChangeFcn properties.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the CreateFcn property.

Uibuttongroup Properties

This section describes all properties useful to uibuttongroup objects and lists valid values. Curly braces {} enclose default values.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackgroundColor</td>
<td>Color of the button group background</td>
</tr>
<tr>
<td>BorderType</td>
<td>Type of border around the button group</td>
</tr>
<tr>
<td>BorderWidth</td>
<td>Width of the button group border in pixels</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Interruption of other callback routines</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Button-press callback routine</td>
</tr>
<tr>
<td>Children</td>
<td>All children of the button group</td>
</tr>
</tbody>
</table>
## Uibuttongroup Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clipping</td>
<td>Clipping of child axes, panels, and button groups to the button group. Does not affect child user interface controls (uicontrol)</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
</tr>
<tr>
<td>FontAngle</td>
<td>Title font angle</td>
</tr>
<tr>
<td>FontName</td>
<td>Title font name</td>
</tr>
<tr>
<td>FontSize</td>
<td>Title font size</td>
</tr>
<tr>
<td>FontUnits</td>
<td>Title font units</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Title font weight</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Title font color and color of 2-D border line</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Handle accessibility from command line and GUIs</td>
</tr>
<tr>
<td>HighlightColor</td>
<td>3-D frame highlight color</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
</tr>
<tr>
<td>Parent</td>
<td>uibuttongroup object’s parent</td>
</tr>
<tr>
<td>Position</td>
<td>Button group position relative to parent figure, panel, or button group</td>
</tr>
<tr>
<td>ResizeFcn</td>
<td>User-specified resize routine</td>
</tr>
<tr>
<td>Selected</td>
<td>Whether object is selected</td>
</tr>
<tr>
<td>SelectedObject</td>
<td>Currently selected uicontrol of style radiobutton or togglebutton</td>
</tr>
<tr>
<td>SelectionChangeFcn</td>
<td>Callback routine executed when the selected radio button or toggle button changes</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Object highlighted when selected</td>
</tr>
</tbody>
</table>
### Uibuttongroup Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShadowColor</td>
<td>3-D frame shadow color</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
</tr>
<tr>
<td>Title</td>
<td>Title string</td>
</tr>
<tr>
<td>TitlePosition</td>
<td>Location of title string in relation to the button group</td>
</tr>
<tr>
<td>Type</td>
<td>Object class</td>
</tr>
<tr>
<td>UICustomMenu</td>
<td>Associate context menu with the button group</td>
</tr>
<tr>
<td>Units</td>
<td>Units used to interpret the position vector</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
</tr>
<tr>
<td>Visible</td>
<td>Button group visibility</td>
</tr>
</tbody>
</table>

**Note** Controls the visibility of a uibuttongroup and of its child axes, uibuttongroups, uipanels, and child uicontrols. Setting it does not change their **Visible** property.

**BackgroundColor**

- **ColorSpec**

  *Color of the uibuttongroup background.* A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the **ColorSpec** reference page for more information on specifying color.

**BorderType**

- none | {etchedin} | etchedout | beveledin | beveledout | line

  *Border of the uibuttongroup area.* Used to define the button group area graphically. Etched and beveled borders provide a 3-D look.
**Uibuttongroup Properties**

Use the `HighlightColor` and `ShadowColor` properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the `ForeColor` property to specify its color.

**BorderWidth**

`integer`

*Width of the button group border.* The width of the button group borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

**BusyAction**

`cancel` | `{queue}`

*Callback routine interruption.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

---

**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. See the `Interruptible` property for information about controlling a callback’s interruptibility.

**ButtonDownFcn**

`string` or `function handle`
Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uibuttongroup. This is useful for implementing actions to interactively modify object properties, such as size and position, when they are clicked on (using the selectmoveresize function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

Children vector of handles

Children of the uibuttongroup. A vector containing the handles of all children of the uibuttongroup. Although a uibuttongroup manages only uicontrols of style radiobutton and togglebutton, its children can be axes, uipanels, uibuttongroups, and other uicontrols. You can use this property to reorder the children.

Clipping {on} | off

Clipping mode. By default, MATLAB clips a uibuttongroup’s child axes, uipanels, and uibuttongroups to the uibuttongroup rectangle. If you set Clipping to off, the axis, uipanel, or uibuttongroup is displayed outside the button group rectangle. This property does not affect child uicontrols which, by default, can display outside the button group rectangle.

CreateFcn string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uibuttongroup object. MATLAB sets all property values for the uibuttongroup before executing the CreateFcn callback so these values are available to
the callback. Within the function, use gcbo to get the handle of the uibuttongroup being created.

Setting this property on an existing uibuttongroup object has no effect.

To define a default CreateFcn callback for all new uibuttongroups you must define the same default for all uipanels. This default applies unless you override it by specifying a different CreateFcn callback when you call uibuttongroup. For example, the code

```matlab
set(0,'DefaultUipanelCreateFcn','set(gcbo,...'
  ''FontName'',''arial'',''FontSize'',12)')
```

creates a default CreateFcn callback that runs whenever you create a new panel or button group. It sets the default font name and font size of the uipanel or uibuttongroup title.

To override this default and create a button group whose FontName and FontSize properties are set to different values, call uibuttongroup with code similar to

```matlab
hpt = uibuttongroup(...,'CreateFcn','set(gcbo,...'
  ''FontName'',''times'',''FontSize'',14)')
```

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uibuttongroup call. In the example above, if instead of redefining the CreateFcn property for this uibuttongroup, you had explicitly set FontSize to 14, the default CreateFcn callback would have set FontSize back to the system dependent default.
Do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object’s RecursionLimit property.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Callback routine executed during object deletion.* A callback routine that executes when you delete the uibuttongroup object (e.g., when you issue a delete command or clear the figure containing the uibuttongroup). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine. The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

**FontAngle**

{normal} | italic | oblique

*Character slant used in the Title.* MATLAB uses this property to select a font from those available on your particular system. Setting this property to italic or oblique selects a slanted version of the font, when it is available on your system.

**FontName**

string

*Font family used in the Title.* The name of the font in which to display the Title. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set
**Uibuttongroup Properties**

FontName to the string FixedWidth. This string value is case insensitive.

```matlab
set(uicontrol_handle,'FontName','FixedWidth')
```

This then uses the value of the root FixedWidthFontName property, which can be set to the appropriate value for a locale from startup.m in the end user's environment. Setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

**FontSize**

integer

*Title font size.* A number specifying the size of the font in which to display the Title, in units determined by the FontUnits property. The default size is system dependent.

**FontUnits**

inches | centimeters | normalized | {points} | pixels

*Title font size units.* Normalized units interpret FontSize as a fraction of the height of the uibuttongroup. When you resize the uibuttongroup, MATLAB modifies the screen FontSize accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

**FontWeight**

light | {normal} | demi | bold

*Weight of characters in the title.* MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

**ForegroundColor**

ColorSpec
Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the ColorSpec reference page for more information on specifying color.

HandleVisibility
{on} | callback | off

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure’s CurrentObject property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when HandleVisibility is on.
- Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.
Note: Uicontrols of style radiobutton and togglebutton that are managed by a uibuttongroup should not be accessed outside the button group. Set the HandleVisibility of such radio buttons and toggle buttons to off or callback to prevent inadvertent access.

You can set the root ShowHiddenHandles property to on to make all handles visible, regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties.

HighlightColor
ColorSpec

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the ColorSpec reference page for more information on specifying color.

Interruptible
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute
If the `Interruptible` property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the waiting callback.

**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure’s `WindowButtonDownFcn` callback routine, or an object’s `ButtonDownFcn` or `Callback` routine is processed according to the rules described above.

**Parent**

`handle`

*`Uibuttongroup parent`*. The handle of the uibuttongroup’s parent figure, uipanel, or uibuttongroup. You can move a uibuttongroup object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

**Position**

`position rectangle`

*Size and location of uibuttongroup relative to parent*. The rectangle defined by this property specifies the size and location
of the button group within the parent figure window, uipanel, or uibuttongroup. Specify Position as

\[ \text{[left bottom width height]} \]

left and bottom are the distance from the lower-left corner of
the parent object to the lower-left corner of the uibuttongroup
object. width and height are the dimensions of the uibuttongroup
rectangle, including the title. All measurements are in units
specified by the Units property.

ResizeFcn
string or function handle

*Resize callback routine.* MATLAB executes this callback routine
whenever a user resizes the uibuttongroup and the figure Resize
property is set to on, or in GUIDE, the Resize behavior option
is set to Other. You can query the uibuttongroup Position
property to determine its new size and position. During execution
of the callback routine, the handle to the figure being resized is
accessible only through the root CallbackObject property, which
you can query using gcbo.

You can use ResizeFcn to maintain a GUI layout that is not
directly supported by the MATLAB Position/Units paradigm.

For example, consider a GUI layout that maintains an object
at a constant height in pixels and attached to the top of the
figure, but always matches the width of the figure. The following
ResizeFcn accomplishes this; it keeps the uicontrol whose Tag is
'StatusBar' 20 pixels high, as wide as the figure, and attached to
the top of the figure. Note the use of the Tag property to retrieve
the uicontrol handle, and the gcbo function to retrieve the figure
handle. Also note the defensive programming regarding figure
Units, which the callback requires to be in pixels in order to work
correctly, but which the callback also restores to their previous
value afterwards.
You can change the figure Position from within the ResizeFcn callback; however, the ResizeFcn is not called again as a result.

Note that the print command can cause the ResizeFcn to be called if the PaperPositionMode property is set to manual and you have defined a resize function. If you do not want your resize function called by print, set the PaperPositionMode to auto.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

Selected

on | off (read only)

Is object selected? This property indicates whether the button group is selected. When this property is on, MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn function to set this property, allowing users to select the object with the mouse.

SelectedObject

scalar handle

Currently selected radio button or toggle button uicontrol in the managed group of components. Use this property to determine the currently selected component or to initialize selection of one of the radio buttons or toggle buttons. By default, SelectedObject is set to the first uicontrol radio button or toggle button that is added. Set it to [] if you want no selection. Note that
SelectionChangeFcn does not execute when this property is set by the user.

**SelectionChangeFcn**

string or function handle

Callback routine executed when the selected radio button or toggle button changes. If this routine is called as a function handle, uibuttongroup passes it two arguments. The first argument, *source*, is the handle of the uibuttongroup. The second argument, *eventdata*, is an event data structure that contains the fields shown in the following table.

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventName</td>
<td>'SelectionChanged'</td>
</tr>
<tr>
<td>oldValue</td>
<td>Handle of the object selected before this event. [] if none was selected.</td>
</tr>
<tr>
<td>newValue</td>
<td>Handle of the currently selected object.</td>
</tr>
</tbody>
</table>

If you have a button group that contains a set of radio buttons and/or toggle buttons and you want an immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group’s SelectionChangeFcn callback function, not in the individual toggle button Callback functions.

If you want another component such as a push button to base its action on the selection, then that component’s Callback callback can get the handle of the selected radio button or toggle button from the button group’s SelectedObject property.
Note  For GUIDE GUIs, hObject contains the handle of the selected radio button or toggle button. See “Examples: Programming GUIDE GUI Components” for more information.

SelectionHighlight
'on' | 'off'

Object highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

ShadowColor
ColorSpec

3-D frame shadow color. ShadowColor is a three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the ColorSpec reference page for more information on specifying color.

Tag
string

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the findobj function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the Tag value 'FormatTb'.

    h = findobj(figurehandles,'Tag','FormatTb')
Title
string

*Title string.* The text displayed in the button group title. You can position the title using the `TitlePosition` property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string in the cell array or padded string matrix is displayed; the rest are ignored. Vertical slash (`'|`) characters are not interpreted as line breaks and instead show up in the text displayed in the uibuttongroup title.

Setting a property value to `default`, `remove`, or `factory` produces the effect described in “Defining Default Values”. To set `Title` to one of these words, you must precede the word with the backslash character. For example,

```matlab
hp = uibuttongroup(...,'Title','\Default');
```

TitlePosition
{lefttop} | centertop | righttop |
leftbottom | centerbottom | rightbottom

*Location of the title.* This property determines the location of the title string, in relation to the uibuttongroup.

Type
string (read-only)

*Object class.* This property identifies the kind of graphics object. For uibuttongroup objects, `Type` is always the string 'uibuttongroup'.

UIContextMenu
handle

*Associate a context menu with a uibuttongroup.* Assign this property the handle of a `Uicontextmenu` object. MATLAB displays
the context menu whenever you right-click the uibuttongroup. Use the uicontextmenu function to create the context menu.

Units
inches | centimeters | {normalized} |
points | pixels | characters

*Units of measurement.* MATLAB uses these units to interpret the Position property. For the button group itself, units are measured from the lower-left corner of its parent figure window, panel, or button group. For children of the button group, they are measured from the lower-left corner of the button group.

- **Normalized** units map the lower-left corner of the button group or figure window to (0,0) and the upper-right corner to (1.0,1.0).
- **Pixels,** inches, centimeters, and points are absolute units (1 point = 1/72 inch).
- **Character** units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

UserData
- matrix

*User-specified data.* Any data you want to associate with the uibuttongroup object. MATLAB does not use this data, but you can access it using set and get.

Visible
- {on} | off
Uibuttongroup visibility. By default, a uibuttongroup object is visible. When set to 'off', the uibuttongroup is not visible, as are all child objects of the button group. When a button group is hidden in this manner, you can still query and set its properties.

**Note** The value of a uibuttongroup’s Visible property determines whether its child components, such as axes, buttons, uipanels, and other uibuttongroups, are visible. However, changing the Visible property of a button group does not change the settings of the Visible property of its child components even though hiding the button group causes them to be hidden.
Purpose
Create context menu

Syntax
handle = uicontextmenu('PropertyName',PropertyValue,...)

Description
handle = uicontextmenu('PropertyName',PropertyValue,...) creates a context menu, which is a menu that appears when the user right-clicks on a graphics object. See the Uicontextmenu Properties reference page for more information.

You create context menu items using the uimenu function. Menu items appear in the order the uimenu statements appear. You associate a context menu with an object using the UIContextMenu property for the object and specifying the context menu's handle as the property value.

Example
These statements define a context menu associated with a line. When the user right clicks or presses Alt+click anywhere on the line, the menu appears. Menu items enable the user to change the line style.

% Create axes and save handle
hax = axes;
% Plot three lines
plot(rand(20,3));
% Define a context menu; it is not attached to anything
hcmenu = uicontextmenu;
% Define callbacks for context menu items that change linestyle
hcb1 = ['set(gco, ''LineStyle'', ''--'')'];
hcb2 = ['set(gco, ''LineStyle'', '':'')'];
hcb3 = ['set(gco, ''LineStyle'', ''-'')'];
% Define the context menu items and install their callbacks
item1 = uimenu(hcmenu, 'Label', 'dashed', 'Callback', hcb1);
item2 = uimenu(hcmenu, 'Label', 'dotted', 'Callback', hcb2);
item3 = uimenu(hcmenu, 'Label', 'solid', 'Callback', hcb3);
% Locate line objects
hlines = findall(hax,'Type','line');
% Attach the context menu to each line
for line = 1:length(hlines)
    set(hlines(line),'uicontextmenu',hcmenu)
When the user right clicks or presses **Alt+click** on the line, the context menu appears, as shown in this figure:

Generally, you need to attach context menus to lines at the time they are plotted in order to be sure that the menus are available to users at once. Therefore, code such as the above could be placed in or called from the callbacks that perform plotting for the GUI.

You should only define callbacks as strings if they need to perform simple actions. For example, if you wanted to add check marks to menu items (using the **Checked** uimenu property) to indicate the current style for a each line, you should define the menu item callbacks as function handles and place the code for them in the GUI’s M-file rather than placing callback strings in the figure.

**See Also**

“Context Menus” in the MATLAB Creating Graphical User Interfaces documentation
uibuttongroup, uicontrol, uimenu, uipanel
Uicontextmenu Properties

Purpose

Describe context menu properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.

- The `set` and `get` functions enable you to set and query the values of properties.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the `CreateFcn` property.

Uicontextmenu Properties

This section lists all properties useful to `uicontextmenu` objects along with valid values and descriptions of their use. Curly braces {} enclose default values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption</td>
</tr>
<tr>
<td>Callback</td>
<td>Control action</td>
</tr>
<tr>
<td>Children</td>
<td>The uimenu defined for the uicontextmenu</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Whether handle is accessible from command line and GUIs</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
</tr>
<tr>
<td>Parent</td>
<td>Uicontextmenu object’s parent</td>
</tr>
</tbody>
</table>
### Uicontextmenu Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>Location of uicontextmenu when Visible is set to on</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
</tr>
<tr>
<td>Visible</td>
<td>Uicontextmenu visibility</td>
</tr>
</tbody>
</table>

**BusyAction**

`cancel | {queue}`

*Callback routine interruption.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. See the `Interruptible` property for information about controlling a callback’s interruptibility.
UiContextMenu Properties

Callback
string

Control action. A routine that executes whenever you right-click an object for which a context menu is defined. The routine executes immediately before the context menu is posted. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

Children
matrix

The uimenu items defined for the uicontextmenu.

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uicontextmenu object. MATLAB sets all property values for the uicontextmenu before executing the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the uicontextmenu being created.

Setting this property on an existing uicontextmenu object has no effect.

You can define a default CreateFcn callback for all new uicontextmenus. This default applies unless you override it by specifying a different CreateFcn callback when you call uicontextmenu. For example, the code

    set(0,'DefaultUiContextMenuCreateFcn','set(gcbo,...
              ''Visible'',''on'')')
creates a default CreateFcn callback that runs whenever you create a new context menu. It sets the default Visible property of a context menu.

To override this default and create a context menu whose Visible property is set to a different value, call uicontextmenu with code similar to

```matlab
hpt = uicontextmenu(...,'CreateFcn','set(gcbo,... ''Visible'',''off'')')
```

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uicontextmenu call. In the example above, if instead of redefining the CreateFcn property for this uicontextmenu, you had explicitly set Visible to off, the default CreateFcn callback would have set Visible back to the default, i.e., on.

Do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object's RecursionLimit property.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Delete uicontextmenu callback routine.* A callback routine that executes when you delete the uicontextmenu object (e.g., when you issue a delete command or clear the figure containing the uicontextmenu). MATLAB executes the routine before destroying
the object’s properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

HandleVisibility
{on} | callback | off

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure’s CurrentObject property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when HandleVisibility is on.
- Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.
You can set the root ShowHiddenHandles property to on to make all handles visible, regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties.

**Interruptible**

{on} | off

*Callback routine interruption mode.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The **Interruptible** property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The **BusyAction** property of the object whose callback is waiting to execute

If the **Interruptible** property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the **Interruptible** property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The **BusyAction** property of the object whose callback is waiting to execute determines what happens to the callback.
**UiContextMenu Properties**

**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure’s `WindowButtonDownFcn` callback routine, or an object’s `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

**Parent**

handle

_UiContextMenu’s parent._ The handle of the uiContextMenu’s parent object, which must be a figure.

**Position**

vector

_UiContextMenu’s position._ A two-element vector that defines the location of a context menu posted by setting the `Visible` property value to on. Specify `Position` as

\[
[x \ y]
\]

where vector elements represent the horizontal and vertical distances in pixels from the bottom left corner of the figure window, panel, or button group to the top left corner of the context menu.

**Tag**

string

_User-specified object label._ The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as
global variables or pass them as arguments between callback routines. You can define Tag as any string.

**Type**

- **string**

*Class of graphics object.* For uicontextmenu objects, **Type** is always the string 'uicontextmenu'.

**UserData**

- **matrix**

*User-specified data.* Any data you want to associate with the uicontextmenu object. MATLAB does not use this data, but you can access it using **set** and **get**.

**Visible**

- **on | {off}**

*Uicontextmenu visibility.* The **Visible** property can be used in two ways:

- Its value indicates whether the context menu is currently posted. While the context menu is posted, the property value is **on**; when the context menu is not posted, its value is **off**.

- Its value can be set to **on** to force the posting of the context menu. Similarly, setting the value to **off** forces the context menu to be removed. When used in this way, the **Position** property determines the location of the posted context menu.
**Purpose**
Create user interface control object

**Syntax**
```matlab
handle = uicontrol('PropertyName', PropertyValue, ...)
handle = uicontrol(parent, 'PropertyName', PropertyValue, ...)
handle = uicontrol
uicontrol(uich)
```

**Description**
`uicontrol` creates a uicontrol graphics objects (user interface controls), which you use to implement graphical user interfaces.

`handle = uicontrol('PropertyName', PropertyValue, ...)` creates a uicontrol and assigns the specified properties and values to it. It assigns the default values to any properties you do not specify. The default uicontrol style is a pushbutton. The default parent is the current figure. See the Uicontrol Properties reference page for more information.

`handle = uicontrol(parent, 'PropertyName', PropertyValue, ...)` creates a uicontrol in the object specified by the handle, parent. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence. `parent` can be the handle of a figure, uipanel, or uibuttongroup.

`handle = uicontrol` creates a pushbutton in the current figure. The `uicontrol` function assigns all properties their default values.

`uicontrol(uich)` gives focus to the uicontrol specified by the handle, `uich`.

When selected, most uicontrol objects perform a predefined action. MATLAB software supports numerous styles of uicontrols, each suited for a different purpose:

- Check boxes
- Editable text fields
- Frames
- List boxes
• Pop-up menus
• Push buttons
• Radio buttons
• Sliders
• Static text labels
• Toggle buttons

For information on using these uicontrols within GUIDE, the MATLAB GUI development environment, see Examples: Programming GUI Components in the MATLAB Creating GUIs documentation

**Specifying the Uicontrol Style**

To create a specific type of uicontrol, set the **Style** property as one of the following strings:

- **'checkbox'** – Check boxes generate an action when selected. These devices are useful when providing the user with a number of independent choices. To activate a check box, click the mouse button on the object. The state of the device is indicated on the display.

- **'edit'** – Editable text fields enable users to enter or modify text values. Use editable text when you want text as input. If Max-Min>1, then multiple lines are allowed. For multi-line edit boxes, a vertical scrollbar enables scrolling, as do the arrow keys.

- **'frame'** – Frames are rectangles that provide a visual enclosure for regions of a figure window. Frames can make a user interface easier to understand by grouping related controls. Frames have no callback routines associated with them. Only other uicontrols can appear within frames.

  Frames are opaque, not transparent, so the order in which you define uicontrols is important in determining whether uicontrols within a frame are covered by the frame or are visible. **Stacking order** determines the order objects are drawn: objects defined first are drawn first; objects defined later are drawn over existing objects. If
you use a frame to enclose objects, you must define the frame before you define the objects.

**Note** Most frames in existing GUIs can now be replaced with panels (uipanel) or button groups (uibuttongroup). GUIDE continues to support frames in those GUIs that contain them, but the frame component does not appear in the GUIDE Layout Editor component palette.

- **'listbox'** – List boxes display a list of items and enable users to select one or more items. The Min and Max properties control the selection mode:
  
  If Max-Min>1, then multiple selection is allowed.
  
  If Max-Min<=1, then only single selection is allowed.
  
  The Value property indicates selected entries and contains the indices into the list of strings; a vector value indicates multiple selections. MATLAB evaluates the list box's callback routine after any mouse button up event that changes the Value property. Therefore, you may need to add a "Done" button to delay action caused by multiple clicks on list items.

  List boxes whose Enable property is on differentiate between single and double left clicks and set the figure SelectionType property to normal or open accordingly before evaluating the list box's Callback property. For such list boxes, Ctrl-left click and Shift-left click also set the figure SelectionType property to normal or open to indicate a single or double click.

- **'popupmenu'** – Pop-up menus (also known as drop-down menus or combo boxes) open to display a list of choices when pressed. When not open, a pop-up menu indicates the current choice. Pop-up menus are useful when you want to provide users with a number of mutually exclusive choices, but do not want to take up the amount of space that a series of radio buttons requires.
• 'pushbutton' – Push buttons generate an action when pressed. To activate a push button, click the mouse button on the push button.

• 'radiobutton' – Radio buttons are similar to check boxes, but are intended to be mutually exclusive within a group of related radio buttons (i.e., only one is in a pressed state at any given time). To activate a radio button, click the mouse button on the object. The state of the device is indicated on the display. Note that your code can implement mutually exclusive behavior for radio buttons.

• 'slider' – Sliders accept numeric input within a specific range by enabling the user to move a sliding bar. Users move the bar by pressing the mouse button and dragging the pointer over the bar, or by clicking in the trough or on an arrow. The location of the bar indicates a numeric value, which is selected by releasing the mouse button. You can set the minimum, maximum, and current values of the slider.

• 'text' – Static text boxes display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively and there is no way to invoke the callback routine associated with it.

• 'togglebutton' – Toggle buttons are controls that execute callbacks when clicked on and indicate their state, either on or off. Toggle buttons are useful for building toolbars.

Remarks

• Adding a uicontrol to a figure removes the figure toolbar when the figure’s Toolbar property is set to 'auto' (which is the default). To prevent this from happening, set the Toolbar property to 'figure'. The user can restore the toolbar by selecting Figure Toolbar from the View menu regardless of this property setting.

• The uicontrol function accepts property name/property value pairs, structures, and cell arrays as input arguments and optionally returns the handle of the created object. You can also set and query property values after creating the object using the set and get functions.
A uicontrol object is a child of a figure, uipanel, or uibuttongroup and therefore does not require an axes to exist when placed in a figure window, uipanel, or uibuttongroup.

When MATLAB is paused and a uicontrol has focus, pressing a keyboard key does not cause MATLAB to resume. Click anywhere outside a uicontrol and then press any key. See the pause function for more information.

Examples

Example 1

The following statement creates a push button that clears the current axes when pressed.

```
h = uicontrol('Style', 'pushbutton', 'String', 'Clear',... 'Position', [20 150 100 70], 'Callback', 'cla');
```

This statement gives focus to the pushbutton.

```
uicontrol(h)
```

Example 2

You can create a uicontrol object that changes figure colormaps by specifying a pop-up menu and supplying an M-file name as the object’s Callback:

```
hpop = uicontrol('Style', 'popup',... 'String', 'hsv|hot|cool|gray',... 'Position', [20 320 100 50],... 'Callback', 'setmap');
```

The above call to uicontrol defines four individual choices in the menu: hsv, hot, cool, and gray. You specify these choices with the String property, separating the choices with the "|" character.

The Callback, in this case setmap, is the name of an M-file that defines a more complicated set of instructions than a single MATLAB command. setmap contains these statements:

```
val = get(hpop,'Value');
```
if val == 1
colormap(hsv)
elseif val == 2
colormap(hot)
elseif val == 3
colormap(cool)
elseif val == 4
colormap(gray)
end

The Value property contains a number that indicates the selected choice. The choices are numbered sequentially from one to four. The setmap M-file can get and then test the contents of the Value property to determine what action to take.

See Also  textwrap, uibuttongroup, uimenu, uipanel
Purpose

Describe user interface control (uicontrol) properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.

- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see “Setting Default Property Values”. You can also set default uicontrol properties on the root and figure levels:

```
set(0,'DefaultUiControlProperty',PropertyValue...)
set(gcf,'DefaultUiControlProperty',PropertyValue...)
```

where `Property` is the name of the uicontrol property whose default value you want to set and `PropertyValue` is the value you are specifying as the default. Use `set` and `get` to access uicontrol properties.

For information on using these uicontrols within GUIDE, the MATLAB GUI development environment, see Programming GUI Components in the MATLAB Creating GUIs documentation.

Uicontrol Properties

This section lists all properties useful to uicontrol objects along with valid values and descriptions of their use. Curly braces {} enclose default values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackgroundColor</td>
<td>Object background color</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Button-press callback routine</td>
</tr>
<tr>
<td>Callback</td>
<td>Control action</td>
</tr>
<tr>
<td>Property</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CData</td>
<td>Truecolor image displayed on the control</td>
</tr>
<tr>
<td>Children</td>
<td>Uicontrol objects have no children</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uicontrol</td>
</tr>
<tr>
<td>Extent</td>
<td>position rectangle (read only)</td>
</tr>
<tr>
<td>FontAngle</td>
<td>Character slant</td>
</tr>
<tr>
<td>FontName</td>
<td>Font family</td>
</tr>
<tr>
<td>FontSize</td>
<td>Font size</td>
</tr>
<tr>
<td>FontUnits</td>
<td>Font size units</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Weight of text characters</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Color of text</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Whether handle is accessible from command line and GUIs</td>
</tr>
<tr>
<td>HitTest</td>
<td>Whether selectable by mouse click</td>
</tr>
<tr>
<td>HorizontalAlignment</td>
<td>Alignment of label string</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
</tr>
<tr>
<td>KeyPressFcn</td>
<td>Key press callback routine</td>
</tr>
<tr>
<td>ListboxTop</td>
<td>Index of top-most string displayed in list box</td>
</tr>
<tr>
<td>Max</td>
<td>Maximum value (depends on uicontrol object)</td>
</tr>
<tr>
<td>Min</td>
<td>Minimum value (depends on uicontrol object)</td>
</tr>
<tr>
<td>Parent</td>
<td>Uicontrol object’s parent</td>
</tr>
</tbody>
</table>
# Uicontrol Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>Size and location of uicontrol object</td>
</tr>
<tr>
<td>Selected</td>
<td>Whether object is selected</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Object highlighted when selected</td>
</tr>
<tr>
<td>SliderStep</td>
<td>Slider step size</td>
</tr>
<tr>
<td>String</td>
<td>Uicontrol object label, also list box and pop-up menu items</td>
</tr>
<tr>
<td>Style</td>
<td>Type of uicontrol object</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
</tr>
<tr>
<td>ToolTipString</td>
<td>Content of object’s tooltip</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
</tr>
<tr>
<td>UICustomMenu</td>
<td>Uicontextmenu object associated with the uicontrol</td>
</tr>
<tr>
<td>Units</td>
<td>Units to interpret position vector</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
</tr>
<tr>
<td>Value</td>
<td>Current value of uicontrol object</td>
</tr>
<tr>
<td>Visible</td>
<td>Uicontrol visibility</td>
</tr>
</tbody>
</table>

**BackgroundColor**

**ColorSpec**

*Object background color.* The color used to fill the uicontrol rectangle. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default color is determined by system settings. See `ColorSpec` for more information on specifying color.
Uicontrol Properties

**Note** On Solaris 2 systems, setting the background color of a slider has no effect.

**BusyAction**

```
cancel | {queue}
```

_Callback routine interruption._ If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback’s interruptibility.

**ButtonDownFcn**

```
string or function handle (GUIDE sets this property)
```

_Button-press callback routine._ A callback routine that can execute when you press a mouse button while the pointer is on or near a `uicontrol`. Specifically:
UiControl Properties

- If the uiControl’s Enable property is set to on, the ButtonDownFcn callback executes when you click the right or left mouse button in a 5-pixel border around the uiControl or when you click the right mouse button on the control itself.

- If the uiControl’s Enable property is set to inactive or off, the ButtonDownFcn executes when you click the right or left mouse button in the 5-pixel border or on the control itself.

This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using selectmoveresize, for example).

Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

To add a ButtonDownFcn callback in GUIDE, select View Callbacks from the Layout Editor View menu, then select ButtonDownFcn. GUIDE sets this property to the appropriate string and adds the callback to the M-file the next time you save the GUI. Alternatively, you can set this property to the string %automatic. The next time you save the GUI, GUIDE sets this property to the appropriate string and adds the callback to the M-file.

Use the Callback property to specify the callback routine that executes when you activate the enabled uiControl (e.g., click a push button).

Callback
string or function handle (GUIDE sets this property)

Control action. A routine that executes whenever you activate the uiControl object (e.g., when you click on a push button or move a slider). Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.
For examples of Callback callbacks for each style of component:

- For GUIDE GUIs, see “Examples: Programming GUIDE GUI Components”.
- For programmatically created GUIs, see “Examples: Programming GUI Components”.

Callback routines defined for static text do not execute because no action is associated with these objects.

To execute the callback routine for an edit text control, type in the desired text and then do one of the following:

- Click another component, the menu bar, or the background of the GUI.
- For a single line editable text box, press Enter.
- For a multiline editable text box, press Ctrl+Enter.

CData matrix

*Truecolor image displayed on control.* A three-dimensional matrix of RGB values that defines a truecolor image displayed on a control, which must be a push button or toggle button. Each value must be between 0.0 and 1.0. Setting CData on a radio button or checkbox will replace the default CData on these controls. The control will continue to work as expected, but its state is not reflected by its appearance when clicked.

For push buttons and toggle buttons, CData overlaps the String. In the case of radio buttons and checkboxes, CData takes precedence over String and, depending on its size, it can displace the text.

Setting CData to [] restores the default CData for radio buttons and checkboxes.
Children
matrix

The empty matrix; uicontrol objects have no children.

Clipping
{on} | off

This property has no effect on uicontrol objects.

CreateFcnc
string or function handle

*Callback routine executed during object creation.* The specified function executes when MATLAB creates a uicontrol object. MATLAB sets all property values for the uicontrol before executing the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the uicontrol being created.

Setting this property on an existing uicontrol object has no effect.

You can define a default CreateFcn callback for all new uicontrols. This default applies unless you override it by specifying a different CreateFcn callback when you call uicontrol. For example, the code

```matlab
set(0,'DefaultUicontrolCreateFcn','set(gcbo,...
    ''BackgroundColor'',''white'')')
```

creates a default CreateFcn callback that runs whenever you create a new uicontrol. It sets the default background color of all new uicontrols.

To override this default and create a uicontrol whose BackgroundColor is set to a different value, call uicontrol with code similar to

```matlab
hpt = uicontrol(...,'CreateFcn','set(gcbo,...
'BackgroundColor', 'blue')

**Note** To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uicontrol` call. In the example above, if instead of redefining the `CreateFcn` property for this `uicontrol`, you had explicitly set `BackgroundColor` to `blue`, the default `CreateFcn` callback would have set `BackgroundColor` back to the default, i.e., `white`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the `uicontrol` object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Delete uicontrol callback routine.* A callback routine that executes when you delete the `uicontrol` object (e.g., when you issue a `delete` command or clear the figure containing the `uicontrol`). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.
Enable
{on} | inactive | off

Enable or disable the uicontrol. This property controls how uicontrols respond to mouse button clicks, including which callback routines execute.

- **on** – The uicontrol is operational (the default).
- **inactive** – The uicontrol is not operational, but looks the same as when Enable is on.
- **off** – The uicontrol is not operational and its image (set by the Cdata property) is grayed out.

When you left-click on a uicontrol whose Enable property is on, MATLAB performs these actions in this order:

1. Sets the figure’s SelectionType property.
2. Executes the uicontrol’s ClickedCallback routine.
3. Does not set the figure’s CurrentPoint property and does not execute either the control’s ButtonDownFcn or the figure’s WindowButtonDownFcn callback.

When you left-click on a uicontrol whose Enable property is off, or when you right-click a uicontrol whose Enable property has any value, MATLAB performs these actions in this order:

1. Sets the figure’s SelectionType property.
2. Sets the figure’s CurrentPoint property.
3. Executes the figure’s WindowButtonDownFcn callback.

Extent
"position rectangle (read only)"

Size of uicontrol character string. A four-element vector that defines the size and position of the character string used to label the uicontrol. It has the form:
Uicontrol Properties

[0,0,width,height]

The first two elements are always zero. width and height are the dimensions of the rectangle. All measurements are in units specified by the Units property.

Since the Extent property is defined in the same units as the uicontrol itself, you can use this property to determine proper sizing for the uicontrol with regard to its label. Do this by

- Defining the String property and selecting the font using the relevant properties.
- Getting the value of the Extent property.
- Defining the width and height of the Position property to be somewhat larger than the width and height of the Extent.

For multiline strings, the Extent rectangle encompasses all the lines of text. For single line strings, the height element of the Extent property returned always indicates the height of a single line, and its width element always indicates the width of the longest line, even if the string wraps when displayed on the control. Edit boxes are considered multilines if Max - Min > 1.

FontAngle

{normal} | italic | oblique

Character slant. MATLAB uses this property to select a font from those available on your particular system. Setting this property to italic or oblique selects a slanted version of the font, when it is available on your system.

FontName

string

Font family. The name of the font in which to display the String. To display and print properly, this must be a font that your system supports. The default font is system dependent.
Note MATLAB GUIs do not support the Marlett and Symbol font families.

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set FontName to the string FixedWidth (this string value is case sensitive):

```
set(uicontrol_handle, 'FontName', 'FixedWidth')
```

This parameter value eliminates the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan). A properly written MATLAB application that needs to use a fixed-width font should set FontName to FixedWidth and rely on the root FixedWidthFontName property to be set correctly in the end user’s environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root FixedWidthFontName property to the appropriate value for that locale from startup.m. Setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

Tip To determine what fonts exist on your system (which can differ from the GUI user’s system), use the uisetfont GUI to select a font and return its name and other characteristics in a MATLAB structure.

FontSize

size in FontUnits
Font size. A number specifying the size of the font in which to display the String, in units determined by the FontUnits property. The default point size is system dependent.

FontUnits
{points} | normalized | inches | centimeters | pixels

Font size units. This property determines the units used by the FontSize property. Normalized units interpret FontSize as a fraction of the height of the uicontrol. When you resize the uicontrol, MATLAB modifies the screen FontSize accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

FontWeight
light | {normal} | demi | bold

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

ForegroundColor
ColorSpec

Color of text. This property determines the color of the text defined for the String property (the uicontrol label). Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default text color is black. See ColorSpec for more information on specifying color.

HandleVisibility
{on} | callback | off

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object
Uicontrol Properties

hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure’s CurrentObject property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when HandleVisibility is on.
- Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root ShowHiddenHandles property to on to make all handles visible, regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties.

__Note__ Radio buttons and toggle buttons that are managed by a uibuttongroup should not be accessed outside the button group. Set the HandleVisibility of such radio buttons and toggle buttons to off to prevent inadvertent access.

HitTest

{on} | off
Selectable by mouse click. This property has no effect on uicontrol objects.

HorizontalAlignment
left | {center} | right

Horizontal alignment of label string. This property determines the justification of the text defined for the String property (the uicontrol label):

- left — Text is left justified with respect to the uicontrol.
- center — Text is centered with respect to the uicontrol.
- right — Text is right justified with respect to the uicontrol.

On Microsoft Windows systems, this property affects only edit and text uicontrols.

Interruptible
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute

If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the drawnow, figure, getframe, pause, or waitfor functions, the function processes
any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

**Note** If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure’s WindowButtonDownFcn callback routine, or an object’s ButtonDownFcn or Callback routine are processed according to the rules described above.

**KeyPressFcn**

string or function handle

*KeyPress callback function.* A callback routine invoked by a key press when the callback’s uicontrol object has focus. Focus is denoted by a border or a dotted border, respectively, in UNIX and Microsoft Windows. If no uicontrol has focus, the figure’s key press callback function, if any, is invoked. KeyPressFcn can be a function handle, the name of an M-file, or any legal MATLAB expression.

If the specified value is the name of an M-file, the callback routine can query the figure’s CurrentCharacter property to determine what particular key was pressed and thereby limit the callback execution to specific keys.
If the specified value is a function handle, the callback routine can retrieve information about the key that was pressed from its event data argument.

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Description</th>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Character interpretation of the key that was pressed.</td>
<td>'a' '=' '' 'A'</td>
</tr>
<tr>
<td>Modifier</td>
<td>Current modifier, such as 'control', or an empty cell array if there is no modifier</td>
<td>{1x0 cell} {1x0 cell} {'shift'} {'shift'}</td>
</tr>
<tr>
<td>Key</td>
<td>Name of the key that was pressed.</td>
<td>'a' 'equal' 'shift' 'a'</td>
</tr>
</tbody>
</table>

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

**ListboxTop**

Scalar

*Index of top-most string displayed in list box.* This property applies only to the `listbox` style of `uicontrol`. It specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. `ListboxTop` is an index into the array of strings defined by the `String` property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

**Max**

Scalar

*Maximum value.* This property specifies the largest value allowed for the `Value` property. Different styles of `uicontrols` interpret `Max` differently:
• Check boxes – Max is the setting of the Value property while the check box is selected.

• Editable text – The Value property does not apply. If Max - Min > 1, then editable text boxes accept multiline input. If Max - Min <= 1, then editable text boxes accept only single line input. The absolute values of Max and Min have no effect on the number of lines an edit box can contain; a multiline edit box can contain any number of lines.

• List boxes – If Max - Min > 1, then list boxes allow multiple item selection. If Max - Min <= 1, then list boxes do not allow multiple item selection. When they do, Value can be a vector of indices.

• Radio buttons – Max is the setting of the Value property when the radio button is selected.

• Sliders – Max is the maximum slider value and must be greater than the Min property. The default is 1.

• Toggle buttons – Max is the value of the Value property when the toggle button is selected. The default is 1.

• Pop-up menus, push buttons, and static text do not use the Max property.

Min

scalar

Minimum value. This property specifies the smallest value allowed for the Value property. Different styles of uicontrols interpret Min differently:

• Check boxes – Min is the setting of the Value property while the check box is not selected.

• Editable text – The Value property does not apply. If Max - Min > 1, then editable text boxes accept multiline input. If Max - Min <= 1, then editable text boxes accept only single line input. The absolute values of Max and Min have no effect on the number of lines.
lines an edit box can contain; a multiline edit box can contain any number of lines.

- List boxes – If \( \text{Max} - \text{Min} > 1 \), then list boxes allow multiple item selection. If \( \text{Max} - \text{Min} \leq 1 \), then list boxes allow only single item selection. When they do, \text{Value} can be a vector of indices.

- Radio buttons – \( \text{Min} \) is the setting of the \text{Value} property when the radio button is not selected.

- Sliders – \( \text{Min} \) is the minimum slider value and must be less than \( \text{Max} \). The default is 0.

- Toggle buttons – \( \text{Min} \) is the value of the \text{Value} property when the toggle button is not selected. The default is 0.

- Pop-up menus, push buttons, and static text do not use the \( \text{Min} \) property.

\textbf{Parent handle}

\emph{Uicontrol parent}. The handle of the uicontrol’s parent object. You can move a uicontrol object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

\textbf{Position position rectangle}

\emph{Size and location of uicontrol}. The rectangle defined by this property specifies the size and location of the control within the parent figure window, uipanel, or uibuttongroup. Specify \text{Position} as

\[ \text{[left bottom width height]} \]

\text{left} and \text{bottom} are the distance from the lower-left corner of the parent object to the lower-left corner of the uicontrol object. \text{width} and \text{height} are the dimensions of the uicontrol rectangle. All measurements are in units specified by the \text{Units} property.
On Microsoft Windows systems, the height of pop-up menus is automatically determined by the size of the font. The value you specify for the height of the Position property has no effect.

The width and height values determine the orientation of sliders. If width is greater than height, then the slider is oriented horizontally. If height is greater than width, then the slider is oriented vertically.

**Note** The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored. The height element of the position vector is not changed.

On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

**Selected**

*on | {off} (read only)*

*Is object selected.* When this property is on, MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

**SelectionHighlight**

*{on} | off*

*Object highlight when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

**SliderStep**

*{min_step max_step}
**Slider step size.** This property controls the amount the slider Value changes when you click the mouse on the arrow button (min_step) or on the slider trough (max_step). Specify SliderStep as a two-element vector; each value must be in the range [0,1], and min_step should be less than max_step. Numbers outside [0 1] can cause the slider not to render or produce unexpected results. The actual step size is a function of the specified SliderStep and the total slider range (Max - Min). The default, [0.01 0.10], provides a 1 percent change for clicks on the arrow button and a 10 percent change for clicks in the trough. and both should be positive numbers less then 1.

For example, if you create the following slider,

```plaintext
uicontrol('Style','slider','Min',1,'Max',7,...
      'Value',2,'SliderStep',[0.1 0.6])
```

clicking on the arrow button moves the indicator by,

```plaintext
0.1*(7-1)
ans =
   0.6000
```

and clicking in the trough moves the indicator by,

```plaintext
0.6*(7-1)
ans =
   3.6000
```

Note that if the specified step size moves the slider to a value outside the range, the indicator moves only to the Max or Min value.

See also the Max, Min, and Value properties.

**String**

String

*UiControl label, list box items, pop-up menu choices.*
For check boxes, editable text, push buttons, radio buttons, static text, and toggle buttons, the text displayed on the object. For list boxes and pop-up menus, the set of entries or items displayed in the object.

**Note** If you specify a numerical value for `String`, MATLAB converts it to `char` but the result may not be what you expect. If you have numerical data, you should first convert it to a string, e.g., using `num2str`, before assigning it to the `String` property.

For uicontrol objects that display only one line of text (check box, push button, radio button, toggle button), if the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash (`|`) characters are not interpreted as line breaks and instead show up in the text displayed in the uicontrol.

For multiple line editable text or static text controls, line breaks occur between each row of the string matrix, and each cell of a cell array of strings. Vertical slash (`|`) characters and `\n` characters are not interpreted as line breaks, and instead show up in the text displayed in the uicontrol.

For multiple items on a list box or pop-up menu, you can specify the items in any of the formats shown in the following table.

<table>
<thead>
<tr>
<th>String Property Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell array of strings</td>
<td><code>{ 'one' 'two' 'three' }</code></td>
</tr>
</tbody>
</table>
### Uicontrol Properties

<table>
<thead>
<tr>
<th><strong>String Property Format</strong></th>
<th><strong>Example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Padded string matrix</td>
<td>['one ';'two ';'three']</td>
</tr>
<tr>
<td>String vector separated by vertical slash (</td>
<td>) characters</td>
</tr>
</tbody>
</table>

If you specify a component width that is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis. Use the Value property to set the index of the initial item selected.

For **check boxes**, **push buttons**, **radio buttons**, **toggle buttons**, and the selected item in **popup menus**, when the specified text is clipped because it is too long for the uicontrol, an ellipsis (...) is appended to the text in the active GUI to indicate that it has been clipped.

For **push buttons** and **toggle buttons**, CData overlaps the String. In the case of **radio buttons** and **checkboxes**, CData takes precedence over String and, depending on its size, can displace the text.

For **editable text**, the String property value is set to the string entered by the user.
UiControl Properties

Reserved Words There are three reserved words: default, remove, factory (case sensitive). If you want to use one of these reserved words in the String property, you must precede it with a backslash ('\') character. For example,

\h = uicontrol('Style','edit','String','\default');

Style

{pushbutton} | togglebutton | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu

Style of uicontrol object to create. The Style property specifies the kind of uicontrol to create. See the uicontrol Description section for information on each type.

Tag

string (GUIDE sets this property)

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

TooltipString

string

Content of tooltip for object. The TooltipString property specifies the text of the tooltip associated with the uicontrol. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

Type

string (read only)
Class of graphics object. For uicontrol objects, Type is always the string 'uicontrol'.

UIContextMenu handle

Associate a context menu with uicontrol. Assign this property the handle of a uicontextmenu object. MATLAB displays the context menu whenever you right-click over the uicontrol. Use the uicontextmenu function to create the context menu.

Units
{pixels} | normalized | inches | centimeters | points | characters (GUIDE default: normalized)

Units of measurement. MATLAB uses these units to interpret the Extent and Position properties. All units are measured from the lower-left corner of the parent object.

- Normalized units map the lower-left corner of the parent object to (0,0) and the upper-right corner to (1.0,1.0).
- pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
- Character units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of Units, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume Units is set to the default value.

UserData matrix
**Uicontrol Properties**

*User-specified data.* Any data you want to associate with the uicontrol object. MATLAB does not use this data, but you can access it using `set` and `get`.

**Value**

scalar or vector

*Current value of uicontrol.* The uicontrol style determines the possible values this property can have:

- Check boxes set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- List boxes set `Value` to a vector of indices corresponding to the selected list entries, where 1 corresponds to the first item in the list.
- Pop-up menus set `Value` to the index of the item selected, where 1 corresponds to the first item in the menu. The Examples section shows how to use the `Value` property to determine which item has been selected.
- Radio buttons set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- Sliders set `Value` to the number indicated by the slider bar.
- Toggle buttons set `Value` to `Max` when they are down (selected) and `Min` when up (not selected).
- Editable text, push buttons, and static text do not set this property.

Set the `Value` property either interactively with the mouse or through a call to the `set` function. The display reflects changes made to `Value`.

**Visible**

{on} | off
UI control visibility. By default, all UI controls are visible. When set to off, the UI control is not visible, but still exists and you can query and set its properties.

**Note** Setting Visible to off for UI controls that are not displayed initially in the GUI, can result in faster startup time for the GUI.
**Purpose**
Open standard dialog box for selecting directory

**Syntax**
```
uigetdir
directory_name = uigetdir
directory_name = uigetdir(start_path)
directory_name = uigetdir(start_path,dialog_title)
```

**Description**
`uigetdir` displays a modal dialog box enabling the user to browse through the directory structure and select a directory or type the name of a directory. If the directory exists, `uigetdir` returns the selected path when the user clicks **OK**. For Microsoft Windows platforms, `uigetdir` opens a dialog box in the base directory (the Windows desktop) with the current directory selected. See “Remarks” on page 2-3875 for information about UNIX and Mac platforms.

**Note** A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

`directory_name = uigetdir` returns the path to the selected directory when the user clicks **OK**. If the user clicks **Cancel** or closes the dialog window, `directory_name` is set to 0.

`directory_name = uigetdir(start_path)` opens a dialog box with the directory specified by `start_path` selected. If `start_path` is a valid directory path, the dialog box opens in the specified directory.

If `start_path` is an empty string (""), the dialog box opens in the current directory. If `start_path` is not a valid directory path, the dialog box opens in the base directory. For Windows, this is the Windows desktop. See “Remarks” on page 2-3875 for information about UNIX and Mac platforms.

`directory_name = uigetdir(start_path,dialog_title)` opens a dialog box with the specified title. On Windows platforms, the
string replaces the default caption inside the dialog box for specifying instructions to the user. The default dialog_title is Select Directory to Open. See “Remarks” on page 2-3875 for information about UNIX and Mac platforms.

**Note** On Windows platforms, users can click the **New Folder** button to add a new directory to the directory structure displayed. Users can also drag and drop existing directories.

**Remarks**

For Windows platforms, the dialog box is similar to those shown in the “Examples” on page 2-3876 below.

For UNIX platforms, uigetdir opens a dialog box in the base directory (the directory from which MATLAB is started) with the current directory selected. The dialog_title string replaces the default title of the dialog box. The dialog box is similar to the one shown in the following figure.
For Mac platforms, `uigetdir` opens a dialog box in the base directory (the current directory) with the current directory open. The `dialog_title` string, if any, is ignored. The dialog box is similar to the one shown in the following figure.

![Dialog Box Example](image)

### Examples

#### Example 1

The following statement displays directories on the C: drive.

```matlab
dname = uigetdir('C:\');
```

The dialog box is shown in the following figure.
If the user selects the directory Desktop, as shown in the figure, and clicks OK, uigetdir returns

\[
dname = C:\WINNT\Profiles\All Users\Desktop
\]

**Example 2**

The following statement uses the `matlabroot` command to display the MATLAB root directory in the dialog box:

```
uigetdir(matlabroot,'MATLAB Root Directory')
```
If the user selects the directory MATLAB6.5/notebook/pc, as shown in the figure, uigetdir returns a string like

C:\MATLAB6.5\notebook\pc

assuming that MATLAB is installed on drive C:\.

See Also

uigetfile, uiputfile
Purpose
Open standard dialog box for retrieving files

Syntax
uigetfile

[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle,DefaultName)
[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',
   selectmode)

Description
uigetfile displays a modal dialog box that lists files in the current
directory and enables the user to select or type the name of a file to be
opened. If the filename is valid and if the file exists, uigetfile returns
the filename when the user clicks Open. Otherwise uigetfile displays
an appropriate error message from which control returns to the dialog
box. The user can then enter another filename or click Cancel. If the
user clicks Cancel or closes the dialog window, uigetfile returns 0.

Successful execution of uigetfile does not open a file; it only returns
the name of an existing file that the user designates.

Note A modal dialog box prevents the user from interacting with other
windows before responding. To block MATLAB program execution, use
the uihat function. For more information about modal dialog boxes,
see WindowStyle in the MATLAB Figure Properties.

[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)
displays only those files with extensions that match FilterSpec.
The uigetfile function appends 'All Files' to the list of file
types.FilterSpec can be a string or a cell array of strings, and can
include the * wildcard.
If FilterSpec is a string that contains a filename, the filename is displayed and selected in the File name field and the file’s extension is used as the default filter.

If FilterSpec is a string, it can include a path. That path can contain '.', '..', or '/'. For example, '../*.m' lists all M-files in the directory above the current directory.

If FilterSpec is a cell array of strings, the first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace standard descriptions in the Files of type field. A description cannot be an empty string. “Example 2” on page 2-3884 and “Example 3” on page 2-3885 illustrate use of a cell array as FilterSpec.

If FilterSpec is not specified, uigetfile uses the default list of file types (i.e., all MATLAB files).

After the user clicks Open and if the filename exists, uigetfile returns the name of the file in FileName and its path in PathName. If the user clicks Cancel or closes the dialog window, FileName and PathName are set to 0.

FilterIndex is the index of the filter selected in the dialog box. Indexing starts at 1. If the user clicks Cancel or closes the dialog window, FilterIndex is set to 0.

[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle) displays a dialog box that has the title DialogTitle. To use the default file types and specify a dialog title, enter

    uigetfile('',DialogTitle)

[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle,DefaultName) displays a dialog box in which the filename specified by DefaultName appears in the File name field. DefaultName can also be a path or a path/filename. In this case, uigetfile opens the dialog box in the directory specified by
the path. See “Example 6” on page 2-3888. Note that you can use '

\texttt{\textquotesingle{.}\textquotesingle{}, \textquotesingle{-}\textquotesingle{}, or \textquotesingle{/}\textquotesingle{} in the \texttt{DefaultName} argument.

If the specified path does not exist, \texttt{uigetfile} opens the dialog box in the current directory.

\begin{verbatim}
[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',selectmode)
\end{verbatim}

sets the multiselect mode to specify if multiple file selection is enabled for the \texttt{uigetfile} dialog. Valid values for \texttt{selectmode} are 'on' and 'off' (default).

If 'MultiSelect' is 'on' and the user selects more than one file in the dialog box, then \texttt{FileName} is a cell array of strings, each of which represents the name of a selected file. Filenames in the cell array are in the sort order native to your platform. Because multiple selections are always in the same directory, \texttt{PathName} is always a string that represents a single directory.

**Remarks**

If the user includes either of the “wildcard” characters ‘*' or ‘?' in a file name, \texttt{uigetfile} does not respond to clicking \texttt{Open}. The dialog box remains open until the user cancels it or removes the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

For Microsoft Windows platforms, the dialog box is the Windows dialog box native to your platform. Because of this, it may differ from those shown in “Examples” on page 2-3883 below.

For UNIX platforms, the dialog box is similar to the one shown in the following figure.
For Mac platforms, the dialog box is similar to the one shown in the following figure.
Examples

Example 1

The following statement displays a dialog box that enables the user to retrieve a file. The statement lists all MATLAB M-files within a selected directory. The name and path of the selected file are returned in FileName and PathName. Note that uigetfile appends All Files(*.*) to the file types when FilterSpec is a string.

\[
\text{[FileName,PathName] = uigetfile('*.m','Select the M-file');}
\]

The dialog box is shown in the following figure.
Example 2

To create a list of file types that appears in the Files of type list box, separate the file extensions with semicolons, as in the following code. Note that `uigetfile` displays a default description for each known file type, such as "Simulink Models" for .mdl files.

    [filename, pathname] = ...    
    uigetfile({'*.*';'*.mdl';'* mat';'*.*'},'File Selector');
Example 3

If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the 'MATLAB Files' description.

```matlab
[filename, pathname] = uigetfile(...
{'.m','.fig','.mat','.mdl'},'MATLAB Files (*.m,.fig,.mat,.mdl)');
  '*.m', 'M-files (*.m)'; ...
  '*.fig', 'Figures (*.fig)'; ...
  '*.mat', 'MAT-files (*.mat)'; ...
  '*.mdl', 'Models (*.mdl)'; ...
  '*.+', 'All Files (*.*)'}, ...
'Pick a file');
```
The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. Note that the first entry of column one contains several extensions, separated by semicolons, all of which are associated with the description 'MATLAB Files (*.m,*.fig,*.mat,*.mdl)'. The code produces the dialog box shown in the following figure.

![Dialog box](image)

**Example 4**

The following code checks for the existence of the file and displays a message about the result of the open operation.

```matlab
[filename, pathname] = uigetfile('*.*', 'Pick an M-file');
```
if isequal(filename,0)
    disp('User selected Cancel')
else
    disp(['User selected', fullfile(pathname, filename)])
end

Example 5

This example creates a list of file types and gives them descriptions that are different from the defaults, then enables multiple file selection. The user can select multiple files by holding down the Shift or Ctrl key and clicking on a file.

    [filename, pathname, filterindex] = uigetfile( ... 
    { '.*.mat', 'MAT-files (*.mat)'; ... 
      '.*.mdl', 'Models (*.mdl)'; ... 
      '.*.*', 'All Files (*.*)'}, ...
'Pick a file', ...
'MultiSelect', 'on');

Example 6

This example uses the `DefaultName` argument to specify a start path and a default filename for the dialog box.

```matlab
uigetfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
           '*.*','All Files' },'mytitle',... 
           'C:\Work\myfile.jpg')
```
See Also  uigetdir, uiputfile
**Purpose**
Open dialog box for retrieving preferences

**Syntax**

```
value = uigetpref(group,pref,title,question,pref_choices)
[val,dlgshown] = uigetpref(...)  
```

**Description**

`value = uigetpref(group,pref,title,question,pref_choices)` returns one of the strings in `pref_choices`, by doing one of the following:

- Prompting the user with a multiple-choice question dialog box
- Returning a previous answer stored in the preferences database

By default, the dialog box is shown, with each choice on a different pushbutton, and with a checkbox controlling whether the returned value should be stored in preferences and automatically reused in subsequent invocations.

If the user checks the checkbox before choosing one of the push buttons, the push button choice is stored in preferences and returned in `value`. Subsequent calls to `uigetpref` detect that the last choice was stored in preferences, and return that choice immediately without displaying the dialog.

If the user does not check the checkbox before choosing a pushbutton, the selected preference is not stored in preferences. Rather, a special value, 'ask', is stored, indicating that subsequent calls to `uigetpref` should display the dialog box.

---

**Note**

`uigetpref` uses the same preference database as `addpref`, `getpref`, `ispref`, `rmpref`, and `setpref`. However, it registers the preferences it sets in a separate list so that it, and `uisetpref`, can distinguish those preferences that are being managed with `uigetpref`.

For preferences registered with `uigetpref`, you can use `setpref` and `uisetpref` to explicitly change preference values to 'ask'.
group and pref define the preference. If the preference does not already exist, uigetpref creates it.

title defines the string displayed in the dialog box titlebar.

question is a descriptive paragraph displayed in the dialog, specified as a string array or cell array of strings. This should contain the question the user is being asked, and should be detailed enough to give the user a clear understanding of their choice and its impact. uigetpref inserts line breaks between rows of the string array, between elements of the cell array of strings, or between '|' or newline characters in the string vector.

pref_choices is either a string, cell array of strings, or '|'-'separated strings specifying the strings to be displayed on the push buttons. Each string element is displayed in a separate push button. The string on the selected pushbutton is returned.

Make pref_choices a 2-by-n cell array of strings if the internal preference values are different from the strings displayed on the pushbuttons. The first row contains the preference strings, and the second row contains the related pushbutton strings. Note that the preference values are returned in value, not the button labels.

[val,dlgshown] = uigetpref(...) returns whether or not the dialog was shown.

Additional arguments can be passed in as parameter-value pairs:

(...'CheckboxState',state) sets the initial state of the checkbox, either checked or unchecked. state can be either 0 (unchecked) or 1 (checked). By default it is 0.

(...'CheckboxString',cbstr) sets the string cbstr on the checkbox. By default it is 'Never show this dialog again'.

(...'HelpString',hstr) sets the string hstr on the help button. By default the string is empty and there is no help button.

(...'HelpFcn',hfcn) sets the callback that is executed when the help button is pressed. By default it is doc('uigetpref'). Note that if there is no 'HelpString' option, a button is not created.
('ExtraOptions',eo) creates extra buttons which are not mapped to any preference settings. eo can be a string or a cell array of strings. By default it is {} and no extra buttons are created. If the user chooses one of these buttons, the dialog is closed and the string is returned in value.

('DefaultButton',dbstr) sets the string value dbstr that is returned if the dialog is closed. By default, it is the first button. Note that dbstr does not have to correspond to a preference or ExtraOption.

**Note** If the preference does not already exist in the preference database, uigetpref creates it. Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

**Examples**

This example creates the following preference dialog for the 'savefigurebeforeclosing' preference in the 'mygraphics' group.

It uses the cell array {'always','never';'Yes','No'} to define the preference values as 'always' and 'never', and their corresponding button labels as 'Yes' and 'No'.

```matlab
[selectedButton,dlgShown]=uigetpref('mygraphics',... % Group
    'savefigurebeforeclosing',... % Preference
    'Closing Figure',... % Window title
    {'Do you want to save your figure before closing?'}
```
'You can save your figure manually by typing `hgsave(gcf)`','

{'always','never';'Yes','No'},...
% Values and button strings

ExtraOptions','Cancel',...
% Additional button

DefaultButton','Cancel',...
% Default choice

HelpString','Help',...
% String for Help button

HelpFcn','doc(''closereq'');')
% Callback for Help button

See Also
addpref, getpref, ispref, rmpref, setpref, uisetpref
Purpose
Open Import Wizard to import data

Syntax
uiimport
uiimport(filename)
uiimport('-file')
uiimport('-pastespecial')
S = uimport(...)

Description
uiimport starts the Import Wizard in the current directory, presenting options to load data from a file or the clipboard.

uiimport(filename) starts the Import Wizard, opening the file specified in filename. The Import Wizard displays a preview of the data in the file.

uiimport('-file') works as above but presents the file selection dialog first.

uiimport('-pastespecial') works as above but presents the clipboard contents first.

S = uimport(...) works as above with resulting variables stored as fields in the struct S.

Note For ASCII data, you must verify that the Import Wizard correctly identified the column delimiter.

See Also
load, importdata, clipboard, fileformats
**Purpose**
Create menus on figure windows

**Syntax**
```
handle = uimenu('PropertyName',PropertyValue,...)
handle = uimenu(parent,'PropertyName',PropertyValue,...)
```

**Description**
`uimenu` creates a hierarchy of menus and submenus that are displayed in the figure window’s menu bar. You also use `uimenu` to create menu items for context menus.

`handle = uimenu('PropertyName',PropertyValue,...)` creates a menu in the current figure’s menu bar using the values of the specified properties and assigns the menu handle to `handle`.

See the Uimenu Properties reference page for more information.

`handle = uimenu(parent,'PropertyName',PropertyValue,...)` creates a submenu of a parent menu or a menu item on a context menu specified by `parent` and assigns the menu handle to `handle`. If `parent` refers to a figure instead of another `uimenu` object or a `uicontextmenu`, MATLAB software creates a new menu on the referenced figure’s menu bar.

**Remarks**
MATLAB adds the new menu to the existing menu bar. If the figure does not have a menu bar, MATLAB creates one. Each menu choice can itself be a menu that displays its submenu when selected. `uimenu` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

The `uimenu` Callback property defines the action taken when you activate the created menu item.

Uimenu only appear in figures whose `Window Style` is normal. If a figure containing `uimenu` children is changed to modal, the `uimenu` children still exist and are contained in the `Children` list of the figure, but are not displayed until the `WindowStyle` is changed to normal.

The value of the figure `MenuBar` property affects the content of the figure menu bar. When `MenuBar` is `figure`, a set of built-in menus precedes any user-created `uimenu`s on the menu bar (MATLAB controls the
built-in menus and their handles are not available to the user). When MenuBar is none, uimenu is the only items on the menu bar (that is, the built-in menus do not appear).

You can set and query property values after creating the menu using set and get.

**Examples**  
This example creates a menu labeled *Workspace* whose choices allow users to create a new figure window, save workspace variables, and exit out of MATLAB. In addition, it defines an accelerator key for the Quit option.

```matlab
f = uimenu('Label','Workspace');
    uimenu(f,'Label','New Figure','Callback','figure');
    uimenu(f,'Label','Save','Callback','save');
    uimenu(f,'Label','Quit','Callback','exit',
        'Separator','on','Accelerator','Q');
```

**See Also**  
uicontrol, uicontextmenu, gcbo, set, get, figure
Uimenu Properties

Purpose
Describe menu properties

Modifying Properties
You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the inspect function at the command line.

- The set and get commands enable you to set and query the values of properties

You can set default Uimenu properties on the root, figure and menu levels:

    set(0,'DefaultUimenuPropertyName',PropertyValue...)
    set(gcf,'DefaultUimenuPropertyName',PropertyValue...)
    set(menu_handle,'DefaultUimenuPropertyName',PropertyValue...)

Where PropertyName is the name of the Uimenu property and PropertyValue is the value you specify as the default for that property.

For more information about changing the default value of property see “Setting Default Property Values”

Uimenu Properties
This section lists all properties useful to uimenu objects along with valid values and instructions for their use. Curly braces {} enclose default values.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator</td>
<td>Keyboard equivalent</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption</td>
</tr>
<tr>
<td>Callback</td>
<td>Control action</td>
</tr>
<tr>
<td>Checked</td>
<td>Menu check indicator</td>
</tr>
<tr>
<td>Children</td>
<td>Handles of submenus</td>
</tr>
</tbody>
</table>
## Uimenu Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uimenu</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Color of text</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Whether handle is accessible from command line and GUIs</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
</tr>
<tr>
<td>Label</td>
<td>Menu label</td>
</tr>
<tr>
<td>Parent</td>
<td>Uimenu object’s parent</td>
</tr>
<tr>
<td>Position</td>
<td>Relative uimenu position</td>
</tr>
<tr>
<td>Separator</td>
<td>Separator line mode</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
</tr>
<tr>
<td>Visible</td>
<td>Uimenu visibility</td>
</tr>
</tbody>
</table>

**Accelerator**

A character.

*Keyboard equivalent.* An alphabetic character specifying the keyboard equivalent for the menu item. This allows users to select a particular menu choice by pressing the specified character in conjunction with another key, instead of selecting the menu item with the mouse. The key sequence is platform specific:
• For Microsoft Windows systems, the sequence is **Ctrl+Accelerator**. These keys are reserved for default menu items: c, v, and x.

• For UNIX systems, the sequence is **Ctrl+Accelerator**. These keys are reserved for default menu items: o, p, s, and w.

You can define an accelerator only for menu items that do not have children menus. Accelerators work only for menu items that directly execute a callback routine, not items that bring up other menus.

Note that the menu item does not have to be displayed (e.g., a submenu) for the accelerator key to work. However, the window focus must be in the figure when the key sequence is entered.

To remove an accelerator, set *Accelerator* to an empty string, ' '.

**BusyAction**

cancel | {queue}

*Callback routine interruption.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of *BusyAction* to decide whether or not to attempt to interrupt the executing callback.

• If the value is **cancel**, the event is discarded and the second callback does not execute.

• If the value is **queue**, and the *Interruptible* property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.
**Uimenu Properties**

**Note** If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. See the `Interruptible` property for information about controlling a callback’s interruptibility.

**Callback**

String or function handle

*Menu action.* A callback routine that executes whenever you select the menu. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

A menu with children (submenus) executes its callback routine before displaying the submenus. A menu without children executes its callback routine when you release the mouse button (i.e., on the button up event).

**Checked**

On | {Off}

*Menu check indicator.* Setting this property to on places a check mark next to the corresponding menu item. Setting it to off removes the check mark. You can use this feature to create menus that indicate the state of a particular option. For example, suppose you have a menu item called *Show axes* that toggles the visibility of an axes between visible and invisible each time the user selects the menu item. If you want a check to appear next to the menu item when the axes are visible, add the following code to the callback for the *Show axes* menu item:

```matlab
if strcmp(get(gcbo, 'Checked'), 'on')
    set(gcbo, 'Checked', 'off');
else
    set(gcbo, 'Checked', 'on');
end
```
set(gcbo, 'Checked', 'on');
end

This changes the value of the Checked property of the menu item from on to off or vice versa each time a user selects the menu item.

Note that there is no formal mechanism for indicating that an unchecked menu item will become checked when selected.

Note This property is ignored for top level and parent menus.

Children

vector of handles

*Handles of submenus.* A vector containing the handles of all children of the uimenu object. The children objects of uimenu objects are other uimenu objects, which function as submenus. You can use this property to reorder the menus.

CreateFcn

string or function handle

*Callback routine executed during object creation.* The specified function executes when MATLAB creates a uimenu object. MATLAB sets all property values for the uimenu before executing the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the uimenu being created.

Setting this property on an existing uimenu object has no effect.

You can define a default CreateFcn callback for all new uimenu objects. This default applies unless you override it by specifying a different CreateFcn callback when you call uimenu. For example, the code
set(0,'DefaultUimenuCreateFcn','set(gcbo,...
    ''Visible'',''on''))

creates a default CreateFcn callback that runs whenever you create a new menu. It sets the default Visible property of a uimenu object.

To override this default and create a menu whose Visible property is set to a different value, call uimenu with code similar to

hpt = uimenu(...,'CreateFcn','set(gcbo,...
    ''Visible'',''off''))

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uimenu call. In the example above, if instead of redefining the CreateFcn property for this uimenu, you had explicitly set Visible to off, the default CreateFcn callback would have set Visible back to the default, i.e., on.

Do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object’s RecursionLimit property.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Delete uimenu callback routine.* A callback routine that executes when you delete the uimenu object (e.g., when you issue a delete command or cause the figure containing the uimenu to reset).
MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which is more simply queried using gcbo.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

Enable
{on} | off

Enable or disable the uimenu. This property controls whether a menu item can be selected. When not enabled (set to off), the menu Label appears dimmed, indicating the user cannot select it.

ForegroundColor
ColorSpec X-Windows only

Color of menu label string. This property determines color of the text defined for the Label property. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default text color is black. See ColorSpec for more information on specifying color.

HandleVisibility
{on} | callback | off

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure’s CurrentObject property. Handles that are hidden are still valid. If you know an object’s
Uimenu Properties

- Handle, you can set and get its properties, and pass it to any function that operates on handles.
- Handles are always visible when HandleVisibility is on.
- Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root ShowHiddenHandles property to on to make all handles visible, regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties.

**Interruptible**
{on} | off

*Callback routine interruption mode.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute
If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the drawnow, figure, getframe, pause, or waitfor functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

**Note** If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure’s WindowButtonDownFcn callback routine, or an object’s ButtonDownFcn or Callback routine are processed according to the rules described above.

**Label**

string

*Menu label.* A string specifying the text label on the menu item. You can specify a mnemonic for the label using the ' & ' character. Except as noted below, the character that follows the ' & ' in the string appears underlined and selects the menu item when you type Alt+ followed by that character while the menu is visible. The ' & ' character is not displayed. To display the ' & ' character in a label, use two ' & ' characters in the string:

'O&pen selection' yields **Open selection**
Uimenu Properties

'Save && Go' yields Save & Go

'Save&&Go' yields Save & Go

'Save& Go' yields Save& Go (the space is not a mnemonic)

There are three reserved words: default, remove, factory (case sensitive). If you want to use one of these reserved words in the Label property, you must precede it with a backslash ('\') character. For example:

'\remove' yields remove

'\default' yields default

'\factory' yields factory

Parent
handle

Uimenu’s parent. The handle of the uimenu’s parent object. The parent of a uimenu object is the figure on whose menu bar it displays, or the uimenu of which it is a submenu. You can move a uimenu object to another figure by setting this property to the handle of the new parent.

Position
scalar

Relative menu position. The value of Position indicates placement on the menu bar or within a menu. Top-level menus are placed from left to right on the menu bar according to the value of their Position property, with 1 representing the left-most position. The individual items within a given menu are placed from top to bottom according to the value of their Position property, with 1 representing the top-most position.
Separator
  on | {off}

Separator line mode. Setting this property to on draws a dividing line above the menu item.

Tag
  string

User-specified object label. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define Tag as any string.

Type
  string (read only)

Class of graphics object. For uimenu objects, Type is always the string 'uimenu'.

UserData
  matrix

User-specified data. Any matrix you want to associate with the uimenu object. MATLAB does not use this data, but you can access it using the set and get commands.

Visible
  {on} | off

Uimenu visibility. By default, all uimenus are visible. When set to off, the uimenu is not visible, but still exists and you can query and set its properties.
**Purpose**  
Convert to unsigned integer

**Syntax**  
I = uint8(X)  
I = uint16(X)  
I = uint32(X)  
I = uint64(X)

**Description**  
I = uint*(X) converts the elements of array X into unsigned integers. X can be any numeric object (such as a double). The results of a uint* operation are shown in the next table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output Range</th>
<th>Output Type</th>
<th>Bytes per Element</th>
<th>Output Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>0 to 255</td>
<td>Unsigned 8-bit integer</td>
<td>1</td>
<td>uint8</td>
</tr>
<tr>
<td>uint16</td>
<td>0 to 65,535</td>
<td>Unsigned 16-bit integer</td>
<td>2</td>
<td>uint16</td>
</tr>
<tr>
<td>uint32</td>
<td>0 to 4,294,967,295</td>
<td>Unsigned 32-bit integer</td>
<td>4</td>
<td>uint32</td>
</tr>
<tr>
<td>uint64</td>
<td>0 to 18,446,744,073,709,551,615</td>
<td>Unsigned 64-bit integer</td>
<td>8</td>
<td>uint64</td>
</tr>
</tbody>
</table>

**double** and **single** values are rounded to the nearest uint* value on conversion. A value of X that is above or below the range for an integer class is mapped to one of the endpoints of the range. For example,

```matlab
uint16(70000)
ans =
65535```

If X is already an unsigned integer of the same class, then uint* has no effect.
You can define or overload your own methods for uint* (as you can for any object) by placing the appropriately named method in an @uint* directory within a directory on your path. Type help datatypes for the names of the methods you can overload.

**Remarks**

Most operations that manipulate arrays without changing their elements are defined for integer values. Examples are reshape, size, the logical and relational operators, subscripted assignment, and subscripted reference.

Some arithmetic operations are defined for integer arrays on interaction with other integer arrays of the same class (e.g., where both operands are uint16). Examples of these operations are +, -, .*, ./, .\ and .^.
If at least one operand is scalar, then *, /, \, and ^ are also defined. Integer arrays may also interact with scalar double variables, including constants, and the result of the operation is an integer array of the same class. Integer arrays saturate on overflow in arithmetic.

**Note** Only the lower order integer data types support math operations. Math operations are not supported for int64 and uint64.

A particularly efficient way to initialize a large array is by specifying the data type (i.e., class name) for the array in the zeros, ones, or eye function. For example, to create a 100-by-100 uint64 array initialized to zero, type

\[
I = \text{zeros}(100, 100, 'uint64');
\]

An easy way to find the range for any MATLAB integer type is to use the intmin and intmax functions as shown here for uint32:

\[
\begin{align*}
\text{intmin}'uint32' & \quad \text{intmax}'uint32' \\
\text{ans} & \quad \text{ans} \\
0 & \quad 4294967295
\end{align*}
\]

**See Also**

double, single, int8, int16, int32, int64, intmax, intmin
uiopen

Purpose
Open file selection dialog box with appropriate file filters

Syntax
uiopen
uiopen('MATLAB')
uiopen('LOAD')
uiopen('FIGURE')
uiopen('SIMULINK')
uiopen('EDITOR')

Description
uiopen displays a modal file selection dialog from which a user can select a file to open. The dialog is the same as the one displayed when you select Open from the File menu in the MATLAB desktop.

Selecting a file in the dialog and clicking Open does the following:

- Gets the file using uigetfile
- Opens the file in the base workspace using the open command

**Note** A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in the MATLAB Figure Properties.

**Note** Only the form uiopen('LOAD') can be compiled into a standalone application. You can create a file selection dialog that can be compiled using uigetfile.

uiopen or uiopen('MATLAB') displays the dialog with the file filter set to all MATLAB files.

uiopen('LOAD') displays the dialog with the file filter set to MAT-files (*.mat).

uiopen('FIGURE') displays the dialog with the file filter set to figure files (*.fig).
uiopen('SIMULINK') displays the dialog with the file filter set to model files (*.mdl).

uiopen('EDITOR') displays the dialog with the file filter set to all MATLAB files except for MAT-files and FIG-files. All files are opened in the MATLAB Editor.

**Examples**

Typing `uiopen('figure')` sets the Files of type field to Figures (*.fig):

![Image of file open dialog with Files of type set to Figures (*.fig)]

**See Also**

uigetfile, uiputfile, uisave
**Purpose**
Create panel container object

**Syntax**

```matlab
h = uipanel('PropertyName1',value1,'PropertyName2',value2,...)
h = uipanel(parent,'PropertyName1',value1,'PropertyName2',value2,...)
```

**Description**

A uipanel groups components. It can contain user interface controls with which the user interacts directly. It can also contain axes, other uipanels, and uibuttongroups. It cannot contain ActiveX controls.

```matlab
h = uipanel('PropertyName1',value1,'PropertyName2',value2,...)
```
creates a uipanel container object in a figure, uipanel, or uibuttongroup. Use the `Parent` property to specify the parent figure, uipanel, or uibuttongroup. If you do not specify a parent, `uipanel` adds the panel to the current figure. If no figure exists, one is created. See the Uipanel Properties reference page for more information.

```matlab
h = uipanel(parent,'PropertyName1',value1,'PropertyName2',value2,...)
```
creates a uipanel in the object specified by the handle, `parent`. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence. `parent` must be a figure, uipanel, or uibuttongroup.

A uipanel object can have axes, uicontrol, uipanel, and uibuttongroup objects as children. For the children of a uipanel, the Position property is interpreted relative to the uipanel. If you move the panel, the children automatically move with it and maintain their positions relative to the panel.

After creating a uipanel object, you can set and query its property values using `set` and `get`.

**Remarks**

If you set the Visible property of a uipanel object to 'off', any child objects it contains (buttons, button groups, axes, etc.) become invisible along with the panel itself. However, doing this does not affect the
settings of the Visible property of any of its child objects, even though all of them remain invisible until the uipanel’s visibility is set to 'on'. uibuttongroup components also behave in this manner.

**Examples**

This example creates a uipanel in a figure, then creates a subpanel in the first panel. Finally, it adds a pushbutton to the subpanel. Both panels use the default Units property value, normalized. Note that default Units for the uicontrol pushbutton is pixels.

```matlab
h = figure;
hp = uipanel('Title','Main Panel','FontSize',12,...
    'BackgroundColor','white','...
    'Position',[.25 .1 .67 .67]);
hsp = uipanel('Parent',hp,'Title','Subpanel','FontSize',12,...
    'Position',[.4 .1 .5 .5]);
hbsp = uicontrol('Parent',hsp,'String','Push here','...
    'Position',[18 18 72 36]);
```

![Figure 1](image)
See Also

hgtransform, uibuttongroup, uicontrol
**Uipanel Properties**

**Purpose**
Describe panel properties

**Modifying Properties**
You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.

- The `set` and `get` functions enable you to set and query the values of properties.

You can set default uipanel properties by typing:

```matlab
set(h,'DefaultUipanelPropertyName',PropertyValue...)
```

Where `h` can be the root handle (0), a figure handle, or a uipanel handle. `PropertyName` is the name of the uipanel property and `PropertyValue` is the value you specify as the default for that property.

**Note** Default properties you set for uipanels also apply to uibuttongroups.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the `CreateFcn` property.

**Uipanel Properties**
This section lists all properties useful to uipanel objects along with valid values and a descriptions of their use. Curly braces {} enclose default values.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackgroundColor</td>
<td>Color of the uipanel background</td>
</tr>
<tr>
<td>BorderType</td>
<td>Type of border around the uipanel area.</td>
</tr>
</tbody>
</table>
## Uipanel Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BorderWidth</td>
<td>Width of the panel border.</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Interruption of other callback routines</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Button-press callback routine</td>
</tr>
<tr>
<td>Children</td>
<td>All children of the uipanel</td>
</tr>
<tr>
<td>Clipping</td>
<td>Clipping of child axes, uipanels, and uibuttongroups to the uipanel. Does not affect child uicontrols.</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
</tr>
<tr>
<td>FontAngle</td>
<td>Title font angle</td>
</tr>
<tr>
<td>FontName</td>
<td>Title font name</td>
</tr>
<tr>
<td>FontSize</td>
<td>Title font size</td>
</tr>
<tr>
<td>FontUnits</td>
<td>Title font units</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Title font weight</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Title font color and/or color of 2-D border line</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Handle accessibility from commandline and GUIs</td>
</tr>
<tr>
<td>HighlightColor</td>
<td>3-D frame highlight color</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
</tr>
<tr>
<td>Parent</td>
<td>Uipanel object’s parent</td>
</tr>
<tr>
<td>Position</td>
<td>Panel position relative to parent figure or uipanel</td>
</tr>
<tr>
<td>ResizeFcn</td>
<td>User-specified resize routine</td>
</tr>
<tr>
<td>Selected</td>
<td>Whether object is selected</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Object highlighted when selected</td>
</tr>
<tr>
<td>ShadowColor</td>
<td>3-D frame shadow color</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
</tr>
<tr>
<td>Title</td>
<td>Title string</td>
</tr>
<tr>
<td>TitlePosition</td>
<td>Location of title string in relation to the panel</td>
</tr>
<tr>
<td>Type</td>
<td>Object class</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Associates uicontextmenu with the uipanel</td>
</tr>
<tr>
<td>Units</td>
<td>Units used to interpret the position vector</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
</tr>
<tr>
<td>Visible</td>
<td>Uipanel visibility.</td>
</tr>
</tbody>
</table>

**Note**  Controls the visibility of a uipanel and of its child axes, uibuttongroups, uipanels, and child uicontrols. Setting it does not change their Visible property.

**BackgroundColor**

- **ColorSpec**

  *Color of the uipanel background.* A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the `ColorSpec` reference page for more information on specifying color.

**BorderType**

- `none` | `{etchedin}` | etchedout | beveledin | beveledout | line
**Uipanel Properties**

*Border of the uipanel area.* Used to define the panel area graphically. Etched and beveled borders provide a 3-D look. Use the HighlightColor and ShadowColor properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the ForegroundColor property to specify its color.

**BorderWidth**
integer

*Width of the panel border.* The width of the panel borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

**BusyAction**
cancel | {queue}

*Callback routine interruption.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

---

**Note** If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. See the Interruptible property for information about controlling a callback’s interruptibility.
ButtonDownFcn

string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uipanel. This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using the selectmoveresize function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

Children

vector of handles

Children of the uipanel. A vector containing the handles of all children of the uipanel. A uipanel object’s children are axes, uipanels, uibuttongroups, and uicontrols. You can use this property to reorder the children.

Clipping

{on} | off

Clipping mode. By default, MATLAB clips a uipanel’s child axes, uipanels, and uibuttongroups to the uipanel rectangle. If you set Clipping to off, the axis, uipanel, or uibuttongroup is displayed outside the panel rectangle. This property does not affect child uicontrols which, by default, can display outside the panel rectangle.

CreateFcn

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uipanel object. MATLAB sets all property values for the uipanel before executing
the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the uipanel being created.

Setting this property on an existing uipanel object has no effect.

You can define a default CreateFcn callback for all new uipanels. This default applies unless you override it by specifying a different CreateFcn callback when you call uipanel. For example, the code

```matlab
set(0,'DefaultUipanelCreateFcn','set(gcbo,...
    ''FontName'',''arial'',''FontSize'','12)')
```

creates a default CreateFcn callback that runs whenever you create a new panel. It sets the default font name and font size of the uipanel title.

**Note** Uibuttongroup takes its default property values from uipanel. Defining a default property for all uipanels defines the same default property for all uibuttongroups.

To override this default and create a panel whose FontName and FontSize properties are set to different values, call uipanel with code similar to

```matlab
hpt = uipanel(...,'CreateFcn','set(gcbo,...
    ''FontName'',''times'',''FontSize'','14)')
```
**Note** To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uipushtool` call. In the example above, if instead of redefining the `CreateFcn` property for this `uipanel`, you had explicitly set `Fontsize` to 14, the default `CreateFcn` callback would have set `FontSize` back to the system dependent default.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object’s `RecursionLimit` property.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Callback routine executed during object deletion.* A callback routine that executes when you delete the `uipanel` object (e.g., when you issue a delete command or clear the figure containing the `uipanel`). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine. The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

**FontAngle**

{normal} | italic | oblique

*Character slant used in the Title.* MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.
FontName
  string

*Font family used in the Title.* The name of the font in which to display the Title. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set FontName to the string FixedWidth (this string value is case insensitive).

```matlab
set(uicontrol_handle,'FontName','FixedWidth')
```

This then uses the value of the root FixedWidthFontName property which can be set to the appropriate value for a locale from startup.m in the end user's environment. Setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

FontSize
  integer

*Title font size.* A number specifying the size of the font in which to display the Title, in units determined by the FontUnits property. The default size is system dependent.

FontUnits
  inches | centimeters | normalized | {points} | pixels

*Title font size units.* Normalized units interpret FontSize as a fraction of the height of the uipanel. When you resize the uipanel, MATLAB modifies the screen FontSize accordingly. pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).

FontWeight
  light | {normal} | demi | bold
**Weight of characters in the title.** MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

**ForegroundColor**

**ColorSpec**

*Color used for title font and 2-D border line.* A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

**HandleVisibility**

{on} | callback | off

*Control access to object’s handle.* This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure’s `CurrentObject` property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is on.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI
UiPanel Properties

(such as evaluating a user-typed string), and so temporarily
hides its own handles during the execution of that function.

You can set the root ShowHiddenHandles property to on to
make all handles visible, regardless of their HandleVisibility
settings. This does not affect the values of the HandleVisibility
properties.

HighlightColor
  ColorSpec

3-D frame highlight color. A three-element RGB vector or one
of the MATLAB predefined names, specifying the highlight
color. See the ColorSpec reference page for more information
on specifying color.

Interruptible
  {on} | off

Callback routine interruption mode. If a callback is executing and
the user triggers an event (such as a mouse click) on an object for
which a callback is defined, that callback attempts to interrupt
the first callback. MATLAB processes the callbacks according to
these factors:

- The Interruptible property of the object whose callback is
  executing
- Whether the executing callback contains drawnow, figure,
  getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is
  waiting to execute

If the Interruptible property of the object whose callback is
executing is on (the default), the callback can be interrupted.
Whenever the callback calls one of the drawnow, figure,
getframe, pause, or waitfor functions, the function processes
Uipanel Properties

any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure’s WindowButtonDownFcn callback routine, or an object’s ButtonDownFcn or Callback routine are processed according to the rules described above.

Parent
handle

*Uipanel parent.* The handle of the uipanel’s parent figure, uipanel, or uibuttongroup. You can move a uipanel object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

Position
position rectangle

*Size and location of uipanel relative to parent.* The rectangle defined by this property specifies the size and location of the panel within the parent figure window, uipanel, or uibuttongroup. Specify Position as

```
[left bottom width height]
```
left and bottom are the distance from the lower-left corner of the parent object to the lower-left corner of the uipanel object. width and height are the dimensions of the uipanel rectangle, including the title. All measurements are in units specified by the Units property.

**ResizeFcn**

string or function handle

*Resize callback routine.* MATLAB executes this callback routine whenever a user resizes the uipanel and the figure Resize property is set to on, or in GUIDE, the Resize behavior option is set to Other. You can query the uipanel Position property to determine its new size and position. During execution of the callback routine, the handle to the figure being resized is accessible only through the root CallbackObject property, which you can query using gcbo.

All axes, uipanel,uitable and uicontrol objects that have their Units set to normalized automatically resize proportionally to the figure. You can define individual resize functions for any such object as needed. For example, you can use ResizeFcn to maintain a GUI layout that is not directly supported by the MATLAB Position/Units paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following ResizeFcn accomplishes this; it keeps the uicontrol whose Tag is 'StatusBar' 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the Tag property to retrieve the uicontrol handle, and the gcbo function to retrieve the figure handle. Also note the defensive programming regarding figure Units, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards.

```matlab
u = findobj('Tag','StatusBar');
```
You can change the figure Position from within a uipanel
ResizeFcn callback; however, the ResizeFcn is not called again
as a result.

A figure’s uipanels resize before the figure itself does. Nested
uipanels resize from inner to outer, with child ResizeFcns being
called before parent ResizeFcns.

Note that the print command can cause the ResizeFcn to be
called if the PaperPositionMode property is set to manual and
you have defined a resize function. If you do not want your resize
function called by print, set the PaperPositionMode to auto.

See “Function Handle Callbacks” for information on how to use
function handles to define the callback function.

See Resize Behavior for information on creating resize functions
using GUIDE.

**Selected**

- **on** | **off** (read only)

*Is object selected?* This property indicates whether the panel is
selected. When this property is on, MATLAB displays selection
handles if the SelectionHighlight property is also on. You
can, for example, define the ButtonDownFcn to set this property,
allowing users to select the object with the mouse.

**SelectionHighlight**

- **{on}** | **off**
Object highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

ShadowColor
ColorSpec

3-D frame shadow color. A three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the ColorSpec reference page for more information on specifying color.

Tag
string

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the findobj function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the Tag value 'FormatTb'.

   h = findobj(figurehandles,'Tag','FormatTb')

Title
string

Title string. The text displayed in the panel title. You can position the title using the TitlePosition property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('\')
characters are not interpreted as line breaks and instead show up in the text displayed in the uipanel title.

Setting a property value to default, remove, or factory produces the effect described in “Defining Default Values”. To set Title to one of these words, you must precede the word with the backslash character. For example,

    hp = uipanel(...,'Title','\Default');

TitlePosition
{lefttop} | centertop | righttop | leftbottom |
centerbottom | rightbottom

Location of the title. This property determines the location of the title string, in relation to the uipanel.

Type
string (read-only)

Object class. This property identifies the kind of graphics object. For uipanel objects, Type is always the string 'uipanel'.

UIContextMenu
handle

Associate a context menu with a uipanel. Assign this property the handle of a Uicontextmenu object. MATLAB displays the context menu whenever you right-click the uipanel. Use the uicontextmenu function to create the context menu.

Units
inches | centimeters | {normalized} | points | pixels |
| characters

Units of measurement. MATLAB uses these units to interpret the Position property. For the panel itself, units are measured from the lower-left corner of the figure window. For children of the panel, they are measured from the lower-left corner of the panel.
Normalized units map the lower-left corner of the panel or figure window to (0,0) and the upper-right corner to (1.0,1.0).

- **pixels, inches, centimeters, and points** are absolute units (1 point = 1/72 inch).

- **Character units** are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of **Units**, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume **Units** is set to the default value.

**UserData**

- **matrix**

*User-specified data.* Any data you want to associate with the **uipanel** object. MATLAB does not use this data, but you can access it using **set** and **get**.

**Visible**

- **{on} | off**

*Uipanel visibility.* By default, a **uipanel** object is visible. When set to 'off', the uipanel is not visible, as are all child objects of the panel. When a panel is hidden in this manner, you can still query and set its properties.

**Note** The value of a uipanel’s **Visible** property determines whether its child components, such as axes, buttons, uibuttongroups, and other uipanels, are visible. However, changing the **Visible** property of a panel does not change the settings of the **Visible** property of its child components even though hiding the panel causes them to be hidden.
**Purpose**  
Create push button on toolbar

**Syntax**

```
hpt = uipushtool('PropertyName1',value1,'PropertyName2', value2,...)
hpt = uipushtool(ht,...)
```

**Description**

```
hpt = uipushtool('PropertyName1',value1,'PropertyName2',value2,...) 
```
creates a push button on the uitoobar at the top of the current figure window, and returns a handle to it. `uipushtool` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function.

Type `get(hpt)` to see a list of uipushtool object properties and their current values. Type `set(hpt)` to see a list of uipushtool object properties that you can set and their legal property values. See the Uipushtool Properties reference page for more information.

```
hpt = uipushtool(ht,...)  
```
creates a button with `ht` as a parent. `ht` must be a uitoobar handle.

**Remarks**

`uipushtool` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

Uipushtools appear in figures whose Window Style is normal or docked. They do not appear in figures whose WindowStyle is modal. If the WindowStyle of a figure containing a uitoobar and its uipushtool children is changed to modal, the uipushtools still exist and are contained in the Children list of the uitoobar, but are not displayed until the figure WindowStyle is changed to normal or docked.

**Examples**

This example creates a uitoobar object and places a uipushtool object on it.

```
h = figure('ToolBar','none')
ht = uitoobar(h)
a = [.20:.05:0.95];
```
b(:, :, 1) = repmat(a, 16, 1)';
b(:, :, 2) = repmat(a, 16, 1);
b(:, :, 3) = repmat(flipdim(a, 2), 16, 1);
hpt = uipushtool(ht, 'CData', b, 'TooltipString', 'Hello')

See Also
get, set, uicontrol, uitoggletool, uitoolbar
Uipushtool Properties

Purpose

Describe push tool properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uipushtool properties by typing:

\[
\text{set}(h, 'DefaultUipushtool\text{PropertyName}', \text{PropertyValue}...) \]

Where \( h \) can be the root handle (0), a figure handle, a uittoolbar handle, or a uipushtool handle. \text{PropertyName} is the name of the Uipushtool property and \text{PropertyValue} is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

Uipushtool Properties

This section lists all properties useful to uipushtool objects along with valid values and a descriptions of their use. Curly braces \{\} enclose default values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeingDeleted</td>
<td>This object is being deleted.</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption.</td>
</tr>
<tr>
<td>CData</td>
<td>Truecolor image displayed on the control.</td>
</tr>
<tr>
<td>ClickedCallback</td>
<td>Control action.</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation.</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Delete uipushtool callback routine.</td>
</tr>
</tbody>
</table>
Uipushtool Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>Enable or disable the uipushtool.</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Control access to object’s handle.</td>
</tr>
<tr>
<td>HitTest</td>
<td>Whether selectable by mouse click</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode.</td>
</tr>
<tr>
<td>Parent</td>
<td>Handle of uipushtool’s parent.</td>
</tr>
<tr>
<td>Separator</td>
<td>Separator line mode</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object label.</td>
</tr>
<tr>
<td>TooltipString</td>
<td>Content of object’s tooltip.</td>
</tr>
<tr>
<td>Type</td>
<td>Object class.</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Uicontextmenu object associated with the uipushtool</td>
</tr>
<tr>
<td>UserData</td>
<td>User specified data.</td>
</tr>
<tr>
<td>Visible</td>
<td>Uipushtool visibility.</td>
</tr>
</tbody>
</table>

BeingDeleted

on | {off} (read only)

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB software sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property). It remains set to on while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object’s BeingDeleted property before acting.
BusyAction
    cancel | {queue}

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

**Note** If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. See the Interruptible property for information about controlling a callback’s interruptibility.

CData
    3-dimensional array

Truecolor image displayed on control. An n-by-m-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your CData array is larger than 16 in the first or second dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.
**Uipushtool Properties**

**ClickedCallback**

string or function handle

*Control action.* A routine that executes when the uipushtool’s `enable` property is set to on, and you press a mouse button while the pointer is on the push tool itself or in a 5-pixel wide border around it.

**CreateFcn**

string or function handle

*Callback routine executed during object creation.* The specified function executes when MATLAB creates a uipushtool object. MATLAB sets all property values for the uipushtool before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the push tool being created.

Setting this property on an existing uipushtool object has no effect.

You can define a default `CreateFcn` callback for all new uipushtools. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uipushtool`. For example, the code

```matlab
imga(:,:,1) = rand(20);
imga(:,:,2) = rand(20);
imga(:,:,3) = rand(20);
set(0,'DefaultUipushtoolCreateFcn','set(gcbo,'Cdata',imga)')
```

creates a default `CreateFcn` callback that runs whenever you create a new push tool. It sets the default image `imga` on the push tool.

To override this default and create a push tool whose `Cdata` property is set to a different image, call `uipushtool` with code similar to
Uipushtool Properties

```matlab
a = [.05:.05:0.95];
imgb(:,:,1) = repmat(a,19,1)';
imgb(:,:,2) = repmat(a,19,1);
imgb(:,:,3) = repmat(flipdim(a,2),19,1);
hpt = uipushtool(...,'CreateFcn','set(gcbo,''CData'',imgb)',...
```

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uipushtool call. In the example above, if instead of redefining the CreateFcn property for this push tool, you had explicitly set CData to imgb, the default CreateFcn callback would have set CData back to imga.

Do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object’s RecursionLimit property.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Callback routine executed during object deletion.* A callback routine that executes when you delete the uipushtool object (e.g., when you call the delete function or cause the figure containing the uipushtool to reset). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.
See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

Enable
{on} | off

Enable or disable the uipushtool. This property controls how uipushtools respond to mouse button clicks, including which callback routines execute.

• on – The uipushtool is operational (the default).
• off – The uipushtool is not operational and its image (set by the Cdata property) is grayed out.

When you left-click on a uipushtool whose Enable property is on, MATLAB performs these actions in this order:

1. Sets the figure’s SelectionType property.
2. Executes the push tool’s ClickedCallback routine.
3. Does not set the figure’s CurrentPoint property and does not execute the figure’s WindowButtonDownFcn callback.

When you left-click on a uipushtool whose Enable property is off, or when you right-click a uipushtool whose Enable property has any value, MATLAB performs these actions in this order:

1. Sets the figure’s SelectionType property.
2. Sets the figure’s CurrentPoint property.
3. Executes the figure’s WindowButtonDownFcn callback.
4. Does not execute the push tool’s ClickedCallback routine.

HandleVisibility
{on} | callback | off

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When
a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure’s CurrentObject property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when HandleVisibility is on.

- Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

- Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root ShowHiddenHandles property to on to make all handles visible, regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties.

**HitTest**

{on} | off

*Selectable by mouse click.* This property has no effect on uipushtool objects.

**Interruptible**

{on} | off
Callback routine interruption mode. If a callback is executing and
the user triggers an event (such as a mouse click) on an object for
which a callback is defined, that callback attempts to interrupt
the first callback. MATLAB processes the callbacks according to
these factors:

- The Interruptible property of the object whose callback is
  executing
- Whether the executing callback contains drawnow, figure,
  getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is
  waiting to execute

If the Interruptible property of the object whose callback is
executing is on (the default), the callback can be interrupted.
Whenever the callback calls one of the drawnow, figure,
getframe, pause, or waitfor functions, the function processes
any events in the event queue, including the waiting callback,
before performing its defined task.

If the Interruptible property of the object whose callback is
executing is off, the callback cannot be interrupted (except by
certain callbacks; see the note below). The BusyAction property
of the object whose callback is waiting to execute determines what
happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn
callback or a figure’s CloseRequest or ResizeFcn callback, it
interrupts an executing callback regardless of the value of that
object’s Interruptible property. The interrupting callback
starts execution at the next drawnow, figure, getframe, pause,
or waitfor statement. A figure’s WindowButtonDownFcn callback
routine, or an object’s ButtonDownFcn or Callback routine are
processed according to the rules described above.
Parent
handle

*Uipushtool parent.* The handle of the uipushtool’s parent toolbar. You can move a uipushtool object to another toolbar by setting this property to the handle of the new parent.

Separator
on | {off}

*Separator line mode.* Setting this property to on draws a dividing line to the left of the uipushtool.

Tag
string

*User-specified object identifier.* The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the `findobj` function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the Tag value 'Copy'.

```matlab
h = findobj(uiprogressbarhandles,'Tag','Copy')
```

TooltipString
string

*Content of tooltip for object.* The TooltipString property specifies the text of the tooltip associated with the uipushtool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

Type
string (read-only)
Object class. This property identifies the kind of graphics object. For uipushtool objects, Type is always the string 'uipushtool'.

**UIContextMenu**

Handle

*Associate a context menu with uicontrol.* This property has no effect on uipushtool objects.

**UserData**

Array

*User specified data.* You can specify UserData as any array you want to associate with the uipushtool object. The object does not use this data, but you can access it using the set and get functions.

**Visible**

{on} | off

*Uipushtool visibility.* By default, all uipushtools are visible. When set to off, the uipushtool is not visible, but still exists and you can query and set its properties.
Purpose
Open standard dialog box for saving files

Syntax
uiputfile
[FileName, PathName, FilterIndex] = uiputfile(FilterSpec)
[FileName, PathName, FilterIndex] = uiputfile(FilterSpec, DialogTitle)
[FileName, PathName, FilterIndex] = uiputfile(FilterSpec, DialogTitle, DefaultName)

Description
uiputfile displays a modal dialog box used to select or specify a file for saving. The dialog box lists the files and directories in the current directory. If the selected or specified filename is valid, it is returned in ans.

If an existing filename is selected or specified, the following warning dialog box is displayed.

The user can select Yes to replace the existing file or No to return to the dialog to select another filename. If the user selects Yes, uiputfile returns the name of the file. If the user selects No, uiputfile returns 0.

Successful execution of uiputfile does not create a file; it only returns the name of a new or existing file that the user designates.

Note  A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the uwait function. For more information about modal dialog boxes, see WindowStyle in the MATLAB Figure Properties.
[FileName, PathName, FilterIndex] = uiputfile(FilterSpec)
displays only those files with extensions that match FilterSpec. The uiputfile function appends 'All Files' to the list of file types. FilterSpec can be a string or a cell array of strings, and can include the * wildcard. For example, '*.m' lists all the MATLAB M-files.

- If FilterSpec is a string that contains a filename, the filename is displayed and selected in the File name field and the file’s extension is used as the default filter.
- If FilterSpec is a string, it can include a path. That path can contain '.', '..', or '/'. For example, '../*.m' lists all M-files in the directory above the current directory.
- If FilterSpec is a cell array of strings, the first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace standard descriptions in the Save as type field. A description cannot be an empty string. “Example 3” on page 2-3949 and “Example 4” on page 2-3950 illustrate use of a cell array as FilterSpec.

If FilterSpec is not specified, uiputfile uses the default list of file types (i.e., all MATLAB files).

After the user clicks Save and if the filename is valid, uiputfile returns the name of the selected file in FileName and its path in PathName. If the user clicks the Cancel button, closes the dialog window, or if the filename is not valid, FileName and PathName are set to 0.

FilterIndex is the index of the filter selected in the dialog box. Indexing starts at 1. If the user clicks the Cancel button, closes the dialog window, or if the file does not exist, FilterIndex is set to 0.

If no output arguments are specified, the filename is returned in ans.

[FileName, PathName, FilterIndex] = uiputfile(FilterSpec,DialogTitle) displays a dialog box that has the title DialogTitle. To use the default file types and specify a dialog title, enter
ui putfile('',DialogTitle)

[FileName,PathName,FilterIndex] =
ui putfile(FilterSpec,DialogTitle,DefaultName) displays a dialog box in which the filename specified by DefaultName appears in the File name field. DefaultName can also be a path or a path/filename. In this case, ui putfile opens the dialog box in the directory specified by the path. See “Example 6” on page 2-3952. Note that you can use '.', '..', or '/' in the DefaultName argument.

If the specified path does not exist, ui putfile opens the dialog box in the current directory.

**Remarks**

If the user includes either of the “wildcard” characters '*' or '?' in a file name, ui putfile does not respond to clicking Save. The dialog box remains open until the user cancels it or removes the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

For Microsoft Windows platforms, the dialog box is the Windows dialog box native to your platform. Because of this, it may differ from those shown in the examples below.

For UNIX platforms, the dialog box is similar to the one shown in the following figure.
For Mac platforms, the dialog box is similar to the one shown in the following figure.
Examples

Example 1

The following statement displays a dialog box titled 'Save file name' with the **Filename** field set to `animinit.m` and the filter set to M-files (*m). Because **FilterSpec** is a string, the filter also includes All Files (*.*),

```
[file,path] = uiputfile('animinit.m','Save file name');
```
Example 2

The following statement displays a dialog box titled 'Save Workspace As' with the filter specifier set to MAT-files.

```matlab
[file,path] = uiputfile('*.mat','Save Workspace As');
```
Example 3

To display several file types in the **Save as type** list box, separate each file extension with a semicolon, as in the following code. Note that `uiputfile` displays a default description for each known file type, such as "Simulink Models" for `.mdl` files.

```matlab
[filename, pathname] = uiputfile(...
    {'*.m'; '*.mdl'; '*.mat'; '*.mat'},...
    'Save as');
```
Example 4

If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the 'MATLAB Files' description.

```matlab
[filename, pathname, filterindex] = uiputfile( ... 
    '*.m;*.fig;*.mat;*.mdl','MATLAB Files (*.m,*.fig,*.mat,*.mdl)'; 
    '*.m', 'M-files (*.m)';... 
    '*.fig','Figures (*.fig)';... 
    '*.mat','MAT-files (*.mat)';... 
    '*.mdl','Models (*.mdl)';...
    '.*','All Files (.*')},...
    'Save as');
```
The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. Note that the first entry of column one contains several extensions, separated by semicolons, all of which are associated with the description 'MATLAB Files (*.m,*.fig,*.mat,*.mdl)'. The code produces the dialog box shown in the following figure.

![Dialog box](image)

**Example 5**

The following code checks for the existence of the file and displays a message about the result of the open operation.

```matlab
[filename, pathname] = uiputfile('*.m','Pick an M-file');
if isequal(filename,0) || isequal(pathname,0)
    disp('User selected Cancel')
else
```

2-3951
disp([ 'User selected', fullfile(pathname, filename) ]) end

Example 6

uiputfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...'*.*','All Files ' },'Save Image',...'C:\Work\newfile.jpg')

See Also uigetdir, uigetfile
Purpose
Resume execution of blocked M-file

Syntax
uiresume(h)

Description
uiresume(h) resumes the M-file execution that uisave suspended.

Remarks
The uisave and uiresume functions block and resume MATLAB program execution. When creating a dialog, you should have a uicontrol component with a callback that calls uiresume or a callback that destroys the dialog box. These are the only methods that resume program execution after the uisave function blocks execution.

When used in conjunction with a modal dialog, uiresume can resume the execution of the M-file that uisave suspended while presenting a dialog box.

Example
This example creates a GUI with a Continue push button. The example calls uisave to block MATLAB execution until uiresume is called. This happens when the user clicks the Continue push button because the push button’s Callback callback, which responds to the click, calls uiresume.

```matlab
f = figure;
h = uicontrol('Position',[20 200 40],'String','Continue',...    'Callback','uiresume(gcf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

gcbf is the handle of the figure that contains the object whose callback is executing.

“Using a Modal Dialog Box to Confirm an Operation” is a more complex example for a GUIDE GUI. See “Icon Editor” for an example for a programmatically created GUI.
See Also
dialog, figure, uicontrol, uimenu, uiwait, waitfor
Purpose
Open standard dialog box for saving workspace variables

Syntax

\begin{verbatim}
uisave
uisave(variables)
uisave(variables,filename)
\end{verbatim}

Description
uisave displays the Save Workspace Variables dialog box for saving workspace variables to a MAT-file, as shown in the figure below. By default, the dialog box opens in your current directory.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{uisave_dialog.png}
\caption{Save Workspace Variables dialog box}
\end{figure}

Note The uisave dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in the MATLAB Figure Properties.

If you type a name in the File name field, such as my_vars, and click Save, the dialog saves all workspace variables in the file my_vars.mat. The default filename is matlab.mat.

uisave(variables) saves only the variables listed in variables. For a single variable, variables can be a string. For more than one variable, variables must be a cell array of strings.
uisave(variables, filename) uses the specified filename as the default **File name** in the Save Workspace Variables dialog box.

**Example**

This example creates workspace variables h and g, and then displays the Save Workspace Variables dialog box in the current directory with the default **File name** set to **var1**.

```matlab
h = 365;
g = 52;
uisave({'h', 'g'}, 'var1');
```

Clicking **Save** stores the workspace variables h and g in the file **var1.mat** in the displayed directory.

**See Also** uigetfile, uiputfile, uiopen
Purpose
Open standard dialog box for setting object’s ColorSpec

Syntax

\[
c = 
\begin{align*}
    & \text{uisetcolor} \\
    & \text{uisetcolor([r g b])} \\
    & \text{uisetcolor}(h) \\
    & \text{uisetcolor}(\ldots, 'dialogTitle')
\end{align*}
\]

Description
\(c = \text{uisetcolor}\) displays a modal color selection dialog appropriate to the platform, and returns the color selected by the user. The dialog box is initialized to white.
\(c = \text{uisetcolor([r g b])}\) displays a dialog box initialized to the specified color, and returns the color selected by the user. \(r\), \(g\), and \(b\) must be values between 0 and 1.
\(c = \text{uisetcolor}(h)\) displays a dialog box initialized to the color of the object specified by handle \(h\), returns the color selected by the user, and applies it to the object. \(h\) must be the handle to an object containing a color property.
\(c = \text{uisetcolor}(\ldots, 'dialogTitle')\) displays a dialog box with the specified title.

If the user presses Cancel from the dialog box, or if any error occurs, the output value is set to the input RGB triple, if provided; otherwise, it is set to 0.

Note
A modal dialog box prevents the user from interacting with other windows before responding. For more information, see WindowStyle in the MATLAB Figure Properties.

See Also
ColorSpec
**Purpose**
Open standard dialog box for setting object’s font characteristics

**Syntax**

```matlab
uisetfont
uisetfont(h)
uisetfont(S)
uisetfont(...,'DialogTitle')
S = uisetfont(...)
```

**Description**

`uisetfont` enables you to change font properties (FontName, FontUnits, FontSize, FontWeight, and FontAngle) for a text, axes, or uicontrol object. The function returns a structure consisting of font properties and values. You can specify an alternate title for the dialog box.

`uisetfont` displays a modal dialog box and returns the selected font properties.

**Note**
A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`uisetfont(h)` displays a modal dialog box, initializing the font property values with the values of those properties for the object whose handle is `h`. Selected font property values are applied to the current object. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(S)` displays a modal dialog box, initializing the font property values with the values defined for the specified structure (S). S must define legal values for one or more of these properties: FontName, FontUnits, FontSize, FontWeight, and FontAngle and the field names must match the property names exactly. If other properties are defined, they are ignored. If a second argument is supplied, it specifies a name for the dialog box.
uisetfont(...,'DialogTitle') displays a modal dialog box with the title DialogTitle and returns the values of the font properties selected in the dialog box.

S = uisetfont(...) returns the properties FontName, FontUnits, FontSize, FontWeight, and FontAngle as fields in a structure. If the user presses Cancel from the dialog box or if an error occurs, the output value is set to 0.

**Example**

These statements create a text object, then display a dialog box (labeled Update Font) that enables you to change the font characteristics:

```matlab
h = text(.5,.5,'Figure Annotation');
uisetfont(h,'Update Font')
```

These statements create two push buttons, then set the font properties of one based on the values set for the other:

```matlab
% Create push button with string ABC
c1 = uicontrol('Style', 'pushbutton', ...
               'Position', [10 10 100 20], 'String', 'ABC');
% Create push button with string XYZ
c2 = uicontrol('Style', 'pushbutton', ...
               'Position', [10 50 100 20], 'String', 'XYZ');
% Display set font dialog box for c1, make selections, & and save to d
d = uisetfont(c1);
% Apply those settings to c2
set(c2, d)
```

**See Also**

axes, text, uicontrol
## uisetpref

**Purpose**  Manage preferences used in uigetpref

**Syntax**  

```matlab
uisetpref('clearall')
```

**Description**  

`uisetpref('clearall')` resets the value of all preferences registered through `uigetpref` to `'ask'`. This causes the dialog box to display when you call `uigetpref`.

**Note**  Use `setpref` to set the value of a particular preference to `'ask'`.

**See Also**  

`setpref`, `uigetpref`
**Purpose**
Reorder visual stacking order of objects

**Syntax**
```
uistack(h)
uistack(h, stackopt)
uistack(h, stackopt, step)
```

**Description**
`uistack(h)` raises the visual stacking order of the objects specified by the handles in `h` by one level (`step` of 1). All handles in `h` must have the same parent.

`uistack(h, stackopt)` moves the objects specified by `h` in the stacking order, where `stackopt` is one of the following:

- `'up'` – moves `h` up one position in the stacking order
- `'down'` – moves `h` down one position in the stacking order
- `'top'` – moves `h` to the top of the current stack
- `'bottom'` – moves `h` to the bottom of the current stack

`uistack(h, stackopt, step)` moves the objects specified by `h` up or down the number of levels specified by `step`.

**Note** In a GUI, `axes` objects are always at a lower level than `uicontrol` objects. You cannot stack an `axes` object on top of a `uicontrol` object.

See “Setting Tab Order” in the MATLAB documentation for information about changing the tab order.

**Example**
The following code moves the child that is third in the stacking order of the figure handle `hObject` down two positions.

```
v = allchild(hObject)
uistack(v(3), 'down', 2)
```
**Purpose**
Create 2-D graphic table GUI component

**Syntax**
uitable
uitable('PropertyName1', value1,'PropertyName2',value2,...)
uitable(parent,...)
handle = uitable(...)

**Description**
uitable creates a 1-by-1 uitable object in the current figure window, using default property values. If no figure exists, a new figure window opens.
uitable('PropertyName1', value1,'PropertyName2',value2,...) creates a uitable object with specified property values. Properties that you do not specify assume the default property values. See the Uitable Properties reference page for information about the available properties.
uitable(parent,...) creates a uitable object as a child of the specified parent handle parent. The parent can be a figure or uipanel handle. If you also specify a different value for the Parent property, the value of the Parent property takes precedence.
handle = uitable(...) creates a uitable object and returns its handle.

After creating a uitable object, you can set and query its property values using the set and get functions.

**Remarks**
Users can change values in a table if the ColumnEditable property is true for the column they attempt to edit. By default, this property is false for all columns. If the column contains pop-up choices, only the current choice is visible (and not the pop-up menu control) when its column cannot be edited.

After editing a value, the edited value is displayed and the CellEditCallback fires when the user does any of the following:

- Types Enter
- Clicks another table cell
• Clicks anywhere else within the table
• Clicks another control or area within the same figure window

However, the CellEditCallback does not fire if the user edits a cell and clicks in another figure window or desktop tool. If the user then returns to the table to edit a different cell, the previous CellEditCallback still does not fire. Even though the table displays the results of the previous edit, the underlying data matrix (the table’s Data property) is not updated to contain the value that appears in the cell.

**Note** If you attempt to create a uitable when running MATLAB on a Linux system without a Java virtual machine (matlab -nojvm) or without a display (matlab nodisplay), no table generates and you receive an error.

**Examples**

**Example 1**

This example creates a table in the current figure. If no figure exists, one is created.

```matlab
    t = uitable;
```
As the table has no content (its Data property is empty), it initially displays no rows or columns. Provide data (a magic square), and set column widths to 25 pixels uniformly to make the entire table visible.

```matlab
set(t,'Data',magic(10))
set(t,'ColumnWidth',{25})
```
The `uitable` `ColumnWidth` property is specified as a cell array. It can contain:

- One number (a width measured in pixels) or the string `auto`
- A cell array containing a list of pixel sizes having up to as many entries as the table has columns

If a list has \( n \) entries, where \( n \) is smaller than the number of columns, it sets the first \( n \) column widths only. You can substitute `auto` for any value in the cell array to have the width of that column calculated automatically.
Example 2

This example creates a table with a 3-by-3 data matrix. This example specifies the column names, parent, and position of the table:

```matlab
f = figure('Position',[100 100 300 150]);
dat = rand(3);
cnames = {'X-Data','Y-Data','Z-Data'};
t = uitable('Data',dat,'ColumnName',cnames,...
    'Parent',f,'Position',[20 20 250 100]);
```

Example 3

This example creates a table with a 3-by-4 array that contains numeric, logical, and string data in the following columns:

- First column (Rate): Numeric, with three decimals (not editable)
- Second column (Amount): Currency (not editable)
- Third column (Available): Check box (editable)
- Fourth column (Fixed/Adj): Pop-up menu with two choices: Fixed and Adjustable (editable)
f = figure('Position', [100 100 400 150]);

dat = {6.125, 456.3457, true, 'Fixed';
      6.75, 510.2342, false, 'Adjustable';
      7, 658.2, false, 'Fixed';};

columnname = {'Rate', 'Amount', 'Available', 'Fixed/Adj'};

columnformat = {'numeric', 'bank', [], {'Fixed' 'Adjustable'}];

columneditable = [false false true true];

t = uitable('Units','normalized','Position',[0.1 0.1 0.9 0.9], 'Data', dat,...
           'ColumnName', columnname,...
           'ColumnFormat', columnformat,...
           'ColumnEditable', columneditable);

For more information about working with uitables, see the following examples in the MATLAB Creating Graphical User Interfaces documentation:

- “GUI to Interactively Explore Data in a Table” (GUIDE example)
- “GUI that Displays and Graphs Tabular Data” (programmatic example)

See Also

figure, format, inspect, uicontrol, uimenu, uipanel
Uitable Properties

Purpose

Describe table properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the inspect function at the command line.

- The set and get functions enable you to set and query the values of properties.

You can set default uitable properties by typing:

```
set(h,'DefaultUitablePropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, or a uitable handle. PropertyName is the name of the uitable property and PropertyValue is the value you specify as the default for that property.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the CreateFcn property.

Uitable Properties

This section lists all properties useful to uitable objects along with valid values and descriptions of their use. In the property descriptions, curly braces {} enclose default values.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackgroundColor</td>
<td>Background color of cells.</td>
</tr>
<tr>
<td>BeingDeleted</td>
<td>This object is being deleted.</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Button-press callback routine</td>
</tr>
<tr>
<td><strong>Property Name</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>CellEditCallback</td>
<td>Callback when data in a cell is changed.</td>
</tr>
<tr>
<td>CellSelectionCallback</td>
<td>Callback when cell is selected</td>
</tr>
<tr>
<td>Children</td>
<td>uitable objects have no children</td>
</tr>
<tr>
<td>Clipping</td>
<td>Does not apply to uitable objects</td>
</tr>
<tr>
<td>ColumnEditable</td>
<td>Determines data in a column as editable</td>
</tr>
<tr>
<td>ColumnFormat</td>
<td>Determines display and editablility of columns</td>
</tr>
<tr>
<td>ColumnName</td>
<td>Column header label</td>
</tr>
<tr>
<td>ColumnWidth</td>
<td>Width of each column in pixels</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine during object creation</td>
</tr>
<tr>
<td>Data</td>
<td>Table data</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine during object deletion</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uitable</td>
</tr>
<tr>
<td>Extent</td>
<td>Size of uitable rectangle</td>
</tr>
<tr>
<td>FontAngle</td>
<td>Character slant of cell content</td>
</tr>
<tr>
<td>FontName</td>
<td>Font family for cell content</td>
</tr>
<tr>
<td>FontSize</td>
<td>Font size of cell content</td>
</tr>
<tr>
<td>FontUnits</td>
<td>Font size units for cell content</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Weight of cell text characters</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Color of text in cells</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Control access to object’s handle</td>
</tr>
<tr>
<td>HitTest</td>
<td>Selectable by mouse click</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
</tr>
<tr>
<td>KeyPressFcn</td>
<td>Key press callback function</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Parent</td>
<td>uitable parent</td>
</tr>
<tr>
<td>Position</td>
<td>Size and location of uitable</td>
</tr>
<tr>
<td>RearrangeableColumn</td>
<td>Location of the column</td>
</tr>
<tr>
<td>RowName</td>
<td>Row header label names</td>
</tr>
<tr>
<td>RowStriping</td>
<td>Color striping of label rows</td>
</tr>
<tr>
<td>Selected</td>
<td>Is object selected?</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Object highlight when selected</td>
</tr>
<tr>
<td>Tag</td>
<td>Use-specified object label</td>
</tr>
<tr>
<td>TooltipString</td>
<td>Content of tooltip for object</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
</tr>
<tr>
<td>UICustomMenu</td>
<td>Associate context menu with uitable</td>
</tr>
<tr>
<td>Units</td>
<td>Units of measurement</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
</tr>
<tr>
<td>Visible</td>
<td>uitable visibility</td>
</tr>
</tbody>
</table>

**BackgroundColor**

1-by-3 or 2-by-3 matrix of RGB triples

*Cell background color.* Color used to fill the uitable cells. Specify as an 1-by-3 or 2-by-3 matrix of RGB triples, such as [.8 .9 .8] or [1 1 .9; .9 1 1]. Each row is an RGB triplet of real numbers between 0.0 and 1.0 that defines one color. (Color names are not allowed.) The default is a 1-by-3 matrix of platform-dependent colors. See `ColorSpec` for information about RGB colors.
Row 2 of the matrix is used only if the RowStriping property is on. The table background is not striped unless both RowStriping is on and the BackgroundColor color matrix has two rows.

**BeingDeleted**

```
on | {off} (read-only)
```

This object is being deleted. The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB software sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property) It remains set to on while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object’s BeingDeleted property before acting.

**BusyAction**

```
cancel | {queue}
```

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the new event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.
Note If the interrupting callback is `DeleteFcn` or `CreateFcn` or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. See the `Interruptible` property for information about controlling a callback’s interruptibility.

**Button-DownFcn**

string or function handle (GUIDE sets this property)

*Button-press callback routine.* A callback routine that can execute when you press a mouse button while the pointer is on or near a uitable. Specifically:

- If the uitable `Enable` property is set to `on`, the `ButtonDownFcn` callback executes when you click the right or left mouse button in a 5-pixel border around the uitable or when you click the right mouse button on the control itself.

- If the uitable `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the right or left mouse button in the 5-pixel border or on the control itself.

This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using the `selectmoveresize` function, for example).

Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

To add a `ButtonDownFcn` callback in GUIDE, select **View Callbacks** from the Layout Editor **View** menu, then select `ButtonDownFcn`. GUIDE sets this property to the appropriate string and adds the callback to the M-file the next time you save the GUI. Alternatively, you can set this property to the string `%automatic`. The next time you save the GUI, GUIDE sets this
property to the appropriate string and adds the callback to the M-file.

**CellEditCallback**

function handle, cell array containing function handle and additional arguments, or string (not recommended)

*Callback to edit user-entered data*

Callback function executed when the user modifies a table cell. It can perform evaluations, validations, or other customizations. If this function is called as a function handle, `uitable` passes it two arguments. The first argument, `source`, is the handle of the `uitable`. The second argument, `eventdata`, is an event data structure that contains the fields shown in the following table. All fields in the event data structure are read only.

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indices</td>
<td>1-by-2 matrix</td>
<td>Row index and column index of the cell the user edited.</td>
</tr>
<tr>
<td>PreviousData</td>
<td>1-by-1 matrix or cell array</td>
<td>Previous data for the changed cell. The default is an empty matrix, <code>[]</code>.</td>
</tr>
<tr>
<td>EditData</td>
<td>String</td>
<td>User-entered string.</td>
</tr>
</tbody>
</table>
## Uitable Properties

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewData</td>
<td>1-by-1 matrix or cell array</td>
<td>Value that uitable wrote to Data. It is either the same as EditData or a converted value, for example, 2 where EditData is '2' and the cell is numeric.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Empty if uitable detected an error in the user-entered data and did not write it to Data.</td>
</tr>
<tr>
<td>Error</td>
<td>String</td>
<td>Error that occurred when uitable tried to convert the EditData string into a value appropriate for Data. For example, uitable could not convert the EditData string consistent with the Column Format property, if any, or the data type for the changed cell.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Empty if uitable wrote the value to Data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If Error is not empty, the CellEditCallback can pass the error string to the user or can attempt to manipulate the data. For example, the string 'pi' would raise an error in a numeric cell but the CellEditCallback could convert it to its numerical equivalent and store it in Data without passing the error to the user.</td>
</tr>
</tbody>
</table>

When a user edits a cell, uitable first attempts to store the user-entered value in Data, converting the value if necessary. It then calls the CellEditCallback and passes it the event data structure. If there is no CellEditCallback and the user-entered data results in an error, the contents of the cell reverts to its previous value and no error is displayed.
Note In order for the CellEditCallback to be issued, after modifying a table cell the user must hit Enter or click somewhere else within the figure containing the table. Editing a cell’s value and then clicking another figure or other window does not save the new value to the data table, and does not fire the CellEditCallback.

CellSelectionCallback
function handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback that executes when cell is selected. Callback function that executes when the user highlights a cell by navigating to it or clicking it. For multiple selection, this callback executes when new cells are added to the selection. The callback includes event data, a structure with one member

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indices</td>
<td>n-by-2 matrix</td>
<td>Row index and column index of the cells the user currently has selected</td>
</tr>
</tbody>
</table>

Once a cell selection has been made, cells within it can be removed one at a time by Ctrl-clicking them.

Children
matrix

The empty matrix; uitable objects have no children.

Clipping
{on} | off

This property has no effect on uitable objects.
**Uitable Properties**

**ColumnEditable**

logical 1-by-n matrix | scalar logical value | {empty matrix ([ ])}

*Determines if column is user-editable.*

Determines if the data can be edited by the end user. Each value in the cell array corresponds to a column. False is default because the developer needs to have control over changes users potentially might make to data.

Specify elements of a logical matrix as true if the data in a column is editable by the user or false if it is not. An empty matrix indicates that no columns are editable.

Columns that contain check boxes or pop-up menus must be editable for the user to manipulate these controls. If a column that contains pop-up menus is not editable, the currently selected choice appears without displaying the pop-up control. The Elements of the ColumnEditable matrix must be in the same order as columns in the Data property. If you do not specify ColumnEditable, the default is an empty matrix ([ ]).

**ColumnFormat**

cell array of strings

*Cell display formatting.* Determines how the data in each column displays and is edited. Elements of the cell array must be in the same order as table columns in the Data property. If you do not want to specify a display format for a particular column, enter [] as a placeholder. If no format is specified for a column, the default display is determined by the data type of the data in the cell. Default ColumnFormat is an empty cell array ({}). In most cases, the default is similar to the command window.

Elements of the cell array must be one of the strings described in the following table.
<table>
<thead>
<tr>
<th><strong>Cell Format</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>'char'</td>
<td>Displays a left-aligned string. To edit, the user types a string that replaces the existing string.</td>
</tr>
<tr>
<td>'logical'</td>
<td>Displays a check box. To edit, the user checks or unchecks the check box. <code>uitable</code> sets the corresponding <code>Data</code> value to <code>true</code> or <code>false</code> accordingly. Initially, the check box is checked if the corresponding <code>Data</code> value would produce <code>true</code> if passed to the <code>logical</code> function, and unchecked otherwise.</td>
</tr>
<tr>
<td>'numeric'</td>
<td>Displays a right-aligned string equivalent to the command window, for numeric data. If the cell <code>Data</code> value is boolean, then 1 or 0 is displayed. If the cell <code>Data</code> value is not numeric and not boolean, then <code>NaN</code> is displayed. To edit, the user can enter any string. This enables a user to enter a value such as 'pi' that can be converted to its numeric equivalent by a <code>CellEditCallback</code>. The <code>uitable</code> function first attempts to convert the user-entered string to a numeric value and store it in <code>Data</code>. It then calls the <code>CellEditCallback</code>. See <code>CellEditCallback</code> for more information.</td>
</tr>
</tbody>
</table>
## Uitable Properties

<table>
<thead>
<tr>
<th>Cell Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-by-n cell array of strings that define a pop-up menu, e.g., {'one', 'two', 'three'}</td>
<td>Displays a pop-up menu. To edit, the user makes a selection from the pop-up menu. <code>uitable</code> sets the corresponding Data value to the selected menu item. The initial values for the pop-up menus in the column are the corresponding strings in Data. These initial values do not have to be items in the pop-up menu. See Example 3 on the <code>uitable</code> reference page.</td>
</tr>
<tr>
<td>Valid string accepted by the format function, e.g., 'short' or 'bank'</td>
<td>Displays the Data value using the specified format. For example, for a two-column table, <code>set(htable, 'ColumnFormat', {'short', 'bank'})</code>.</td>
</tr>
</tbody>
</table>

In some cases, you may need to insert an appropriate column in Data. If Data is a numerical or logical matrix, you must first convert it to a cell array using the `mat2cell` function.

### Data and ColumnFormat

When you create a table, you must specify value of Data. The Data property dictates what type of data can exist in any given cell. By default, the value of the Data also dictates the display of the cell to the end user, unless you specify a different format using the ColumnFormat property.
ColumnFormat controls the presentation of the Data to the end user. Therefore, if you specify a ColumnFormat of char (or pick Text from the Table Property Editor), you are asking the table to display the Data associated with that column as a string. For example, if the Data for a particular column is numeric, and you specify the ColumnFormat as char, then the display of the numeric data will be left-aligned.
If your column is editable and the user enters a number, the number will be left-aligned. However, if the user enters a text string, the table displays a NaN.

Another possible scenario is that the value Data is char and you set the ColumnFormat to be a pop-up menu. Here, if the value of the Data in the cell matches one of the pop-up menu choices you define in ColumnFormat, then the Data is shown in the cell. If it does not match, then the cell defaults to display the first option from the choices you specify in ColumnFormat. Similarly, if Data is numeric or logical with the ColumnFormat as pop-up menu, if the Data value in the cell does not match any of the choices you specify in ColumnFormat, the cell defaults to display the first option in the pop-menu choice.

This table describes how Data values correspond with your ColumnFormat when the columns are editable.

<table>
<thead>
<tr>
<th>ColumnFormat Selections</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>logical</td>
</tr>
<tr>
<td><strong>Data Type</strong></td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td><strong>char</strong></td>
</tr>
<tr>
<td><strong>logical</strong></td>
</tr>
</tbody>
</table>

**Note** If you have defined `CellEditCallback`, this warning will not be thrown.
If you get a mismatch error, you have the following options:

- Change the ColumnFormat or value of Data to match.
- Implement the CellEditCallback to handle custom data conversion.

**ColumnName**

1–by-*n* cell array of strings | {'numbered'} | empty matrix ([ ])

*Column heading names.* Each element of the cell array is the name of a column. Multiline column names can be expressed as a string vector separated by vertical slash (|) characters, e.g., 'Standard|Deviation'

For sequentially numbered column headings starting with 1, specify ColumnName as 'numbered'. This is the default.

To remove the column headings, specify ColumnName as the empty matrix ([ ]).

The number of columns in the table is the larger of ColumnName and the number of columns in the Data property matrix or cell array.

**ColumnWidth**

1–by-*n* cell array or 'auto'

*Column widths.* The width of each column in units of pixels. Column widths are always specified in pixels; they do not obey the Units property. Each column in the cell array corresponds to a column in the uitable. By default, the width of the column name, as specified in ColumnName, along with some other factors, is used to determine the width of a column. If ColumnWidth is a cell array and the width of a column is set to 'auto' or if auto is selected for that column in the Property Inspector GUI for columns, the column width defaults to a size determined by the table. The table decides the default size using a number of factors, including the ColumnName and the minimum column size.
To default all column widths in an existing table, use

```matlab
set(uitable_handle,'ColumnWidth','auto')
```

To default some column widths but not others, use a cell array containing a mixture of pixel values and 'auto'. For example,

```matlab
set(uitable_handle,'ColumnWidth',{64 'auto' 40 40 'auto' 72})
```

CreateFcn

string or function handle

*Callback routine executed during object creation.* The specified function executes when MATLAB creates a `uitable` object. MATLAB sets all property values for the table before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the table being created.

Setting this property on an existing `uitable` object has no effect.

You can define a default `CreateFcn` callback for all new `uitable`s. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uitable`. For example, the code

```matlab
set(0,'DefaultUitableCreateFcn','set(gcbo,...
   ''BackgroundColor'',''blue'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new `uitable`. It sets the default background color of all new `uitable`s.

To override this default and create a `uitable` whose `BackgroundColor` is set to a different value, call `uitable` with code similar to

```matlab
hpt = uitable(...,'CreateFcn','set(gcbo,...
   ''BackgroundColor'',''white'')')
```
**Note** To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uitable` call. In the example above, if instead of redefining the `CreateFcn` property for this `uitable`, you had explicitly set `BackgroundColor` to white, the default `CreateFcn` callback would have set `BackgroundColor` back to the default, i.e., blue.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**Data**

matrix or cell array of numeric, logical, or character data

*Data content of uitable.* The matrix or cell array must be 2-dimensional. A cell array can mix data types.

Use `get` and `set` to modify `Data`. For example,

```matlab
    data = get(tablehandle,'Data')
    data(event.indices(1),event.indices(2)) = pi();
    set(tablehandle,'Data',data);
```

See `CellEditCallback` for information about the event data structure. See `ColumnFormat` for information about specifying the data display format.
The number of rows in the table is the larger of RowName and the number of rows in Data. The number of columns in the table is the larger of ColumnName and the number of columns in Data.

**DeleteFcn**

string or function handle

*Delete uitable callback routine.* A callback routine that executes when you delete the uitable object (e.g., when you issue a `delete` command or clear the figure containing the uitable). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root CallbackObject property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define a callback function.

**Enable**

{on} | inactive | off

*Enable or disable the uitable.* This property determines how uitables respond to mouse button clicks, including which callback routines execute.

- on – The uitable is operational (the default).
- inactive – The uitable is not operational, but looks the same as when Enable is on.
- off – The uitable is not operational and its image is grayed out.

When you left-click on a uitable whose `Enable` property is on, MATLAB performs these actions in this order:

1. Sets the figure’s `SelectionType` property.
2 Executes theuitable’s CellSelectionCallback routine (but only for table cells, not header cells). Row and column indices of the cells the user selects continuously update the Indices field in the eventdata passed to the callback.

3 Does not set the figure’s CurrentPoint property and does not execute either the table’s ButtonDownFcn or the figure’s WindowButtonDownFcn callback.

When you left-click on auitable whose Enable property is off, or when you right-click auitable whose Enable property has any value, MATLAB performs these actions in this order:

1 Sets the figure’s SelectionType property.
2 Sets the figure’s CurrentPoint property.
3 Executes the figure’s WindowButtonDownFcn callback.

Extent
position rectangle (read only)

Size ofuitable rectangle. A four-element vector of the form [0,0,width,height] that contains the calculated values of the largest extent of the table based on the current Data, RowNames and ColumnNames property values. Calculation depends on column and row widths, when they are available. The calculated extent can be larger than the figure.

The first two elements are always zero. width and height are the dimensions of the rectangle. All measurements are in units specified by the Units property.

When theuitable’s Units property is set to 'Normalized', its Extent is measured relative to the figure, regardless of whether the table is contained in (parented to) a uipanel or not.

You can use this property to determine proper sizing for the uitable with respect to its content. Do this by setting the width
and height of the uitable Position property to the width and height of the Extent property. However, doing this can cause the table to extend beyond the right or top edge of the figure and/or its uipanel parent, if any, for tables with large extents.

FontAngle
{normal} | italic | oblique

*Character slant of cell content.* MATLAB uses this property to select a font from those available on your particular system. Setting this property to italic or oblique selects a slanted version of the font, when it is available on your system.

FontName
string

*Font family for cell content.* The name of the font in which to display cell content. To display and print properly, this must be a font that your system supports. The default font is system dependent.

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set FontName to the string FixedWidth (this string value is case sensitive):

```
set(uitable_handle,'FontName','FixedWidth')
```

This parameter value eliminates the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan). A properly written MATLAB application that needs to use a fixed-width font should set FontName to FixedWidth and rely on the root FixedWidthFontName property to be set correctly in the end user’s environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root FixedWidthFontName
property to the appropriate value for that locale from `startup.m`. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

**FontSize**

size in `FontUnits`

*Font size for cell contents.* A number specifying the size of the font in which to display cell contents, in units determined by the `FontUnits` property. The default point size is system dependent. If `FontUnits` is set to `normalized`, `FontSize` is a number between 0 and 1.

**FontUnits**

{points} | normalized | inches | centimeters | pixels

*Font size units for cell contents.* This property determines the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the `uitable`. When you resize the `uitable`, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

**FontWeight**

light | {normal} | demi | bold

*Weight of cell text characters.* MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

**ForegroundColor**

1-by-3 matrix of RGB triples or a color name

*Color of text in cells.* Determines the color of the text defined for cell contents. Text in all cells share the current color. Specify as a 1-by-3 matrix of RGB triples, such as [0 0 .8] or as a color name.
The default is a 1-by-3 matrix of platform-dependent colors. See `ColorSpec` for information about specifying RGB colors.

**HandleVisibility**

{on} | callback | off

*Control access to object’s handle.* This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure’s `CurrentObject` property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is on.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

**HitTest**

{on} | off
Selectable by mouse click. When HitTest is off, the ButtonDownFcn callback does not execute.

Interruptible
{on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. The MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute

If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the drawnow, figure, getframe, pause, or waitfor functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.
**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure’s `WindowButtonDownFcn` callback routine, or an object’s `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

**KeyPressFcn**

string or function handle

*Key press callback function.* A callback routine invoked by a key press when the callback’s uitable object has focus. Focus is denoted by a border or a dotted border, respectively, in UNIX and Microsoft Windows. If no uitable has focus, the figure’s key press callback function, if any, is invoked. `KeyPressFcn` can be a function handle, the name of an M-file, or any legal MATLAB expression.

If the specified value is the name of an M-file, the callback routine can query the figure’s `CurrentCharacter` property to determine what particular key was pressed and thereby limit the callback execution to specific keys.

If the specified value is a function handle, the callback routine can retrieve information about the key that was pressed from its event data structure argument.

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Description</th>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Character interpretation of the key that was pressed.</td>
<td><code>'a'</code></td>
</tr>
</tbody>
</table>
Uitable Properties

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Description</th>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifier</td>
<td>Current modifier, such as 'control', or an empty cell array if there is no modifier</td>
<td>{1x0 cell} {1x0 cell} {'shift'} {'shift'}</td>
</tr>
<tr>
<td>Key</td>
<td>Name of the key that was pressed.</td>
<td>'a' 'equal' 'shift' 'a'</td>
</tr>
</tbody>
</table>

The uitable KeyPressFcn callback executes for all keystrokes, including arrow keys or when a user edits cell content.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

Parent

handle

Uitable parent. The handle of the uitable’s parent object. You can move a uitable object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

Position

position rectangle

Size and location of uitable. The rectangle defined by this property specifies the size and location of the table within the parent figure window, ui, or uibuttongroup. Specify Position as a 4–element vector:

[left bottom width height]

left and bottom are the distance from the lower-left corner of the parent object to the lower-left corner of the uitable object. width and height are the dimensions of the uitable rectangle. All measurements are in units specified by the Units property.
Note If you are specifying both Units and Position in the same call to uitable, specify Units first if you want Position to be interpreted using those units.

RearrangeableColumn
  on | {off}

This object can be rearranged. The RearrangeableColumn property provides a mechanism that you can use to reorder the columns in the table. All columns are rearrangeable when this property is turned on. MATLAB software sets the RearrangeableColumn property to off by default.

When this property is on, the user of a table can move any column of data (but not the row labels) at a time left or right to reorder it by clicking and dragging its header. Rearranging columns does not affect the ordering of columns in the table’s Data, only the user’s view of it.

RowName
  1–by-n cell array of strings | {'numbered'} | empty matrix ([])

Row heading names. Each element of the cell array is the name of a row. Row names are restricted to one line of text.

For sequentially numbered row headings starting with 1, specify RowName as ‘numbered’. This is the default.

To remove the row headings, specify RowName as the empty matrix ([]).

The number of rows in the table is the larger of RowName and the number of rows in the Data property matrix or cell array.

RowStriping
  {on} | off
**Table Properties**

*Color striping of table rows.* When `RowStriping` is on, the background of consecutive rows of the table display in the pair of colors that the `BackgroundColor` color matrix specifies. The first color matrix row applies to odd-numbered rows, and the second to even-numbered rows. If the `BackgroundColor` matrix has only one row, it is applied to all rows (that is, no striping occurs).

When `RowStriping` is off, the first color specified for `BackgroundColor` is applied to all rows.

**Selected**

*on | {off}*

*Is object selected.* When this property is on, MATLAB displays selection handles if the `SelectionHighlight` property is also on. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**SelectionHighlight**

*{on} | off*

*Object highlight when selected.* When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is off, MATLAB does not draw the handles.

**Tag**

*string (GUIDE sets this property)*

*User-specified object label.* The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

**TooltipString**

*string*
Content of tooltip for object. The TooltipString property specifies the text of the tooltip associated with the uitable. When the user moves the mouse pointer over the table and leaves it there, the tooltip is displayed.

Type
string (read only)

Class of graphics object. For uitable objects, Type is always the string 'uitable'.

UIContextMenu handle

Associate a context menu with uitable. Assign this property the handle of a uicontextmenu object. MATLAB displays the context menu whenever you right-click over the uitable. Use the uicontextmenu function to create the context menu.

Units
{pixels} | normalized | inches | centimeters | points | characters (GUIDE default: normalized)

Units of measurement. MATLAB uses these units to interpret the Extent and Position properties. All units are measured from the lower-left corner of the parent object.

- Normalized units map the lower-left corner of the parent object to (0,0) and the upper-right corner to (1.0,1.0).
- pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
- Character units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of Units, it is good practice to return it to its default value after completing your computation so as not
to affect other functions that assume Units is set to the default value.

**UserData**
matrix

*User-specified data.* Any data you want to associate with the uitable object. MATLAB does not use this data, but you can access it using set and get.

**Visible**
{on} | off

*Uitable visibility.* By default, all uitables are visible. When set to off, the uitable is not visible, but still exists and you can query and set its properties.

---

**Note** Setting Visible to off for uitables that are not displayed initially in the GUI, can result in faster startup time for the GUI.
Purpose
Create toggle button on toolbar

Syntax
htt = uitoggletool('PropertyName1',value1,'PropertyName2',value2,...)
htt = uitoggletool(ht,...)

Description
htt = uitoggletool('PropertyName1',value1,'PropertyName2',value2,...)
creates a toggle button on the uitoolbar at the top of the current figure
window, and returns a handle to it. uitoggletool assigns the
specified property values, and assigns default values to the
remaining properties. You can change the property values at a
later time using the set function.

Type get(htt) to see a list of uitoggletool object properties and their
current values. Type set(htt) to see a list of uitoggletool object
properties you can set and legal property values. See the Uitoggletool
Properties reference page for more information.

htt = uitoggletool(ht,...) creates a button with ht as a parent.
ht must be a uitoolbar handle.

Remarks
uitoggletool accepts property name/property value pairs, as well as
structures and cell arrays of properties as input arguments.

Toggle tools appear in figures whose Window Style is normal or docked.
They do not appear in figures whose WindowStyle is modal. If the
WindowStyle property of a figure containing a uitoobar and its toggle
tool children is changed to modal, the toggle tools still exist and are
contained in the Children list of the tool bar, but are not displayed
until the WindowStyle is changed to normal or docked.

Examples
This example creates a uitoobar object and places a uitoggletool
object on it.

    h = figure('ToolBar','none');
    ht = uitoobar(h);
    a = rand(16,16,3);
 UIToggleTool

```matlab
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello');
```

See Also

get, set, uicontrol, uipushtool, uitoobar
Purpose

Describe toggle tool properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the inspect function at the command line.
- The set and get functions enable you to set and query the values of properties.

You can set default Uitoggletool properties by typing:

```
set(h,'DefaultUitoggletoolPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, a uutoolbar handle, or a uitoggletool handle. PropertyName is the name of the Uitoggletool property and PropertyValue is the value you specify as the default for that property.

For more information about changing the default value of a property see “Setting Default Property Values”.

Properties

This section lists all properties useful to uitoggletool objects along with valid values and a descriptions of their use. Curly braces {} enclose default values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeingDeleted</td>
<td>This object is being deleted.</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption.</td>
</tr>
<tr>
<td>CData</td>
<td>Truecolor image displayed on the toggle tool.</td>
</tr>
<tr>
<td>ClickedCallback</td>
<td>Control action independent of the toggle tool position.</td>
</tr>
</tbody>
</table>
# Uitoggletool Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation.</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion.</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uitoggletool.</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Control access to object’s handle.</td>
</tr>
<tr>
<td>HitTest</td>
<td>Whether selectable by mouse click</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode.</td>
</tr>
<tr>
<td>OffCallback</td>
<td>Control action when toggle tool is set to the off position.</td>
</tr>
<tr>
<td>OnCallback</td>
<td>Control action when toggle tool is set to the on position.</td>
</tr>
<tr>
<td>Parent</td>
<td>Handle of uitoggletool’s parent toolbar.</td>
</tr>
<tr>
<td>Separator</td>
<td>Separator line mode.</td>
</tr>
<tr>
<td>State</td>
<td>Uitoggletool state.</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object label.</td>
</tr>
<tr>
<td>TooltipsString</td>
<td>Content of object’s tooltip.</td>
</tr>
<tr>
<td>Type</td>
<td>Object class.</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Uicontextmenu object associated with the uitoggletool</td>
</tr>
<tr>
<td>UserData</td>
<td>User specified data.</td>
</tr>
<tr>
<td>Visible</td>
<td>Uitoggletool visibility.</td>
</tr>
</tbody>
</table>

**BeingDeleted**

```
|-being deleted| (read only) |
```

This object is being deleted. The **BeingDeleted** property provides a mechanism that you can use to determine if objects are
in the process of being deleted. MATLAB software sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property) It remains set to on while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object’s BeingDeleted property before acting.

**BusyAction**

```
cancel | {queue}
```

*Callback routine interruption.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

**Note** If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. See the Interruptible property for information about controlling a callback’s interruptibility.

**CData**

```
3-dimensional array
```
Truecolor image displayed on control. An n-by-m-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your CData array is larger than 16 in the first or second dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

ClickedCallback
string or function handle

Control action independent of the toggle tool position. A routine that executes after either the OnCallback routine or OffCallback routine runs to completion. The uitoggletool’s Enable property must be set to on.

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uitoggletool object. MATLAB sets all property values for the uitoggletool before executing the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the toggle tool being created.

Setting this property on an existing uitoggletool object has no effect.

You can define a default CreateFcn callback for all new uitoggletools. This default applies unless you override it by specifying a different CreateFcn callback when you call uitoggletool. For example, the statement,

```
set(0,'DefaultUitoggletoolCreateFcn','...
   'set(gcbo,'''Enable''',''off''))'
```
Uitoggletool Properties

creates a default CreateFcn callback that runs whenever you create a new toggle tool. It sets the toggle tool Enable property to off.

To override this default and create a toggle tool whose Enable property is set to on, you could call uitoggletool with code similar to

```matlab
htt = uitoggletool(...,'CreateFcn',...
    'set(gcbo,''Enable'',''on''),...)
```

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uitoggletool call. In the example above, if instead of redefining the CreateFcn property for this toggle tool, you had explicitly set Enable to on, the default CreateFcn callback would have set CData back to off.

Do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object’s RecursionLimit property.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Callback routine executed during object deletion.* A callback routine that executes when you delete the uitoggletool object (e.g., when you call the delete function or cause the figure containing the uitoggletool to reset). MATLAB executes the routine before
destroying the object’s properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

Enable
{on} | off

Enable or disable the uitoggletool. This property controls how uitoggletools respond to mouse button clicks, including which callback routines execute.

• on – The uitoggletool is operational (the default).
• off – The uitoggletool is not operational and its image (set by the Cdata property) is grayed out.

When you left-click on a uitoggletool whose Enable property is on, MATLAB performs these actions in this order:
1 Sets the figure’s SelectionType property.
2 Executes the toggle tool’s ClickedCallback routine.
3 Does not set the figure’s CurrentPoint property and does not execute the figure’s WindowButtonDownFcn callback.

When you left-click on a uitoggletool whose Enable property is off, or when you right-click a uitoggletool whose Enable property has any value, MATLAB performs these actions in this order:
4 Sets the figure’s SelectionType property.
5 Sets the figure’s CurrentPoint property.
6 Executes the figure’s WindowButtonDownFcn callback.
7 Does not execute the toggle tool’s OnCallback, OffCallback, or ClickedCallback routines.

**HandleVisibility**
{on} | callback | off

*Control access to object’s handle.* This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure’s `CurrentObject` property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is on.

- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

**HitTest**
{on} | off
**Uitoggletool Properties**

*Selectable by mouse click.* This property has no effect on uitoggletool objects.

**Interruptible**

{on} | off

*Callback routine interruption mode.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The **Interruptible** property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The **BusyAction** property of the object whose callback is waiting to execute

If the **Interruptible** property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the **Interruptible** property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below).
**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement.

**OffCallback**

string or function handle

*Control action.* A routine that executes if the uitoggletool’s `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `off`.
- The toggle tool is set to the `off` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

**OnCallback**

string or function handle

*Control action.* A routine that executes if the uitoggletool’s `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `on`.
- The toggle tool is set to the `on` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.
Parent

handle

_Uitoggletool parent_. The handle of the uitoggletool’s parent toolbar. You can move a uitoggletool object to another toolbar by setting this property to the handle of the new parent.

Separator

on | {off}

_Separator line mode_. Setting this property to on draws a dividing line to left of the uitoggletool.

State

on | {off}

_Uitoggletool state_. When the state is on, the toggle tool appears in the down, or pressed, position. When the state is off, it appears in the up position. Changing the state causes the appropriate OnCallback or OffCallback routine to run.

Tag

string

_User-specified object identifier_. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the _findobj_ function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the Tag value 'Bold'.

```
h = findobj(uuitoolbarhandles, 'Tag', 'Bold')
```

TooltipString

string
**Uitoggletool Properties**

*Content of tooltip for object.* The `TooltipString` property specifies the text of the tooltip associated with the uitoggletool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

**Type**

`string` (read-only)

Object class. This property identifies the kind of graphics object. For uitoggletool objects, Type is always the string 'uitoggletool'.

**UIContextMenu**

`handle`

*Associate a context menu with uicontrol.* This property has no effect on uitoggletool objects.

**UserData**

`array`

*User specified data.* You can specify `UserData` as any array you want to associate with the uitoggletool object. The object does not use this data, but you can access it using the `set` and `get` functions.

**Visible**

{on} | off

*Uitoggletool visibility.* By default, all uitoggletools are visible. When set to off, the uitoggletool is not visible, but still exists and you can query and set its properties.
Purpose
Create toolbar on figure

Syntax
ht = uitoolbar('PropertyName1',value1,'PropertyName2',value2,...)
ht = uitoolbar(h,...)

Description
ht = uitoolbar('PropertyName1',value1,'PropertyName2',value2,...)
creates an empty toolbar at the top of the current figure window, and
returns a handle to it. uitoolbar assigns the specified property values,
and assigns default values to the remaining properties. You can change
the property values at a later time using the set function.

Type get(ht) to see a list of uitoolbar object properties and their
current values. Type set(ht) to see a list of uitoolbar object properties
that you can set and legal property values. See the Uitoolbar Properties
reference page for more information.

ht = uitoolbar(h,...) creates a toolbar with h as a parent. h must
be a figure handle.

Remarks
uitoolbar accepts property name/property value pairs, as well as
structures and cell arrays of properties as input arguments.

Uitoolbars appear in figures whose Window Style is normal or docked.
They do not appear in figures whose WindowStyle is modal. If the
WindowStyle property of a figure containing a uitoolbar is changed to
modal, the uitoolbar still exists and is contained in the Children list
of the figure, but is not displayed until the WindowStyle is changed
to normal or docked.

Example
This example creates a figure with no toolbar, then adds a toolbar to it.

    h = figure('ToolBar','none')
    ht = uitoolbar(h)
For more information on using the menus and toolbar in a MATLAB figure window, see the online MATLAB Graphics documentation.

**See Also**

set, get, uicontrol, uipushtool, uitoggletool
Purpose

Describe toolbar properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.

- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uitoolbar properties by typing:

```matlab
set(h,'DefaultUitoolbarPropertyName',PropertyValue...)
```

Where `h` can be the root handle (0), a figure handle, or a uicontrol handle. `PropertyName` is the name of the Uitoolbar property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

Uitoolbar Properties

This section lists all properties useful to `uicontrol` objects along with valid values and a descriptions of their use. Curly braces `{}` enclose default values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeingDeleted</td>
<td>This object is being deleted.</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption.</td>
</tr>
<tr>
<td>Children</td>
<td>Handles of uicontrol's children.</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation.</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion.</td>
</tr>
</tbody>
</table>
Uitoolbar Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandleVisibility</td>
<td>Control access to object’s handle.</td>
</tr>
<tr>
<td>HitTest</td>
<td>Whether selectable by mouse click</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode.</td>
</tr>
<tr>
<td>Parent</td>
<td>Handle of uitoolbar’s parent.</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier.</td>
</tr>
<tr>
<td>Type</td>
<td>Object class.</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Uicontextmenu object associated with the uitoolbar</td>
</tr>
<tr>
<td>UserData</td>
<td>User specified data.</td>
</tr>
<tr>
<td>Visible</td>
<td>Uitoolbar visibility.</td>
</tr>
</tbody>
</table>

**BeingDeleted**

| on | {off} (read-only) |

*This object is being deleted.* The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB software sets the BeingDeleted property to on when the object’s delete function callback is called (see the DeleteFcn property) It remains set to on while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object’s BeingDeleted property before acting.

**BusyAction**

| cancel | {queue} |

*Callback routine interruption.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new
event uses the value of BusyAction to decide whether or not to attempt to interrupt the executing callback.

- If the value is cancel, the event is discarded and the second callback does not execute.
- If the value is queue, and the Interruptible property of the first callback is on, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. See the Interruptible property for information about controlling a callback’s interruptibility.

Children
vector of handles

Handles of tools on the toolbar. A vector containing the handles of all children of the uiotoolbar object, in the order in which they appear on the toolbar. The children objects of uitoolbars are uipushbuttons and uitoggles. You can use this property to reorder the children.

CreateFcn
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uiotoolbar object. MATLAB sets all property values for the uiotoolbar before executing the CreateFcn callback so these values are available to the callback. Within the function, use gcbo to get the handle of the toolbar being created.
Setting this property on an existing uitoobar object has no effect.

You can define a default CreateFcn callback for all new uitoobars. This default applies unless you override it by specifying a different CreateFcn callback when you call uitoobar. For example, the statement,

```matlab
set(0,'DefaultUitoolbarCreateFcn',...
    'set(gcbo,''Visibility'',''off'')')
```

creates a default CreateFcn callback that runs whenever you create a new toolbar. It sets the toolbar visibility to off.

To override this default and create a toolbar whose Visibility property is set to on, you could call uitoobar with a call similar to

```matlab
ht = uitoobar(...,'CreateFcn',...
    'set(gcbo,''Visibility'',''on'')',...)
```

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uitoobar call. In the example above, if instead of redefining the CreateFcn property for this toolbar, you had explicitly set Visibility to on, the default CreateFcn callback would have set Visibility back to off.

Do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object's RecursionLimit property.
See Function Handle Callbacks for information on how to use function handles to define a callback function.

**DeleteFcn**

string or function handle

*Callback routine executed during object deletion.* A callback function that executes when the uitoobar object is deleted (e.g., when you call the `delete` function or cause the figure containing the uitoobar to reset). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

Within the function, use `gcbo` to get the handle of the toolbar being deleted.

**HandleVisibility**

{on} | callback | off

*Control access to object’s handle.* This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure’s `CurrentObject` property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is on.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
• Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root ShowHiddenHandles property to on to make all handles visible, regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties.

HitTest
{on} | off

*Selectable by mouse click.* This property has no effect on uitoobar objects.

Interruptible
{on} | off

*Callback routine interruption mode.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

• The Interruptible property of the object whose callback is executing

• Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements

• The BusyAction property of the object whose callback is waiting to execute

If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the drawnow, figure, getframe, pause, or waitfor functions, the function processes
any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

**Note** If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure’s CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object’s Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure’s WindowButtonDownFcn callback routine, or an object’s ButtonDownFcn or Callback routine are processed according to the rules described above.

**Parent**

handle

*Uitoolbar parent.* The handle of the uitoobar’s parent figure. You can move a uitoobar object to another figure by setting this property to the handle of the new parent.

**Tag**

string

*User-specified object identifier.* The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the findobj function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the
handles of all children (of the specified figures) that have the Tag value 'FormatTb'.

\[ h = \text{findobj(figurehandles,'Tag','FormatTb')} \]

**Type**

string (read-only)

Object class. This property identifies the kind of graphics object. For uicontrol objects, **Type** is always the string 'uicontrol'.

**UIContextMenu**

handle

*Associate a context menu with uicontrol.* This property has no effect on uicontrol objects.

**UserData**

array

*User specified data.* You can specify **UserData** as any array you want to associate with the uicontrol object. The object does not use this data, but you can access it using the **set** and **get** functions.

**Visible**

{on} | off

*Uitoolbar visibility.* By default, all uitoolbars are visible. When set to off, the uitoolbar is not visible, but still exists and you can query and set its properties.
uiwait

Purpose
Block execution and wait for resume

Syntax
uiwait
uiwait(h)
uiwait(h,timeout)

Description
uiwait blocks execution until uiresume is called or the current figure is deleted. This syntax is the same as uiwait(gcf).

uiwait(h) blocks execution until uiresume is called or the figure h is deleted.

uiwait(h,timeout) blocks execution until uiresume is called, the figure h is deleted, or timeout seconds elapse.

Remarks
The uiwait and uiresume functions block and resume MATLAB and Simulink program execution. uiwait also blocks the execution of Simulink models. The functions pause (with no argument) and waitfor also block execution in this manner. uiwait is a convenient way to use the waitfor command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the M-file that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, uiwait can block the execution of the M-file and restrict user interaction to the dialog only.

Example
This example creates a GUI with a Continue push button. The example calls uiwait to block MATLAB execution until uiresume is called. This happens when the user clicks the Continue push button because the push button’s Callback callback, which responds to the click, calls uiresume.

```matlab
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',
                'Callback','uiresume(gcf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
```
close(f);

gcbf is the handle of the figure that contains the object whose callback is executing.

“Using a Modal Dialog Box to Confirm an Operation” is a more complex example for a GUIDE GUI. See “Icon Editor” for an example for a programmatically created GUI.

See Also
dialog, figure, uicontrol, uimenu, uiresume, waitfor
**Purpose**
 Undo previous checkout from source control system (UNIX platforms)

**GUI Alternatives**
 As an alternative to the undocheckout function, select Source Control > Undo Checkout in the File menu of the Editor, Simulink software, or Stateflow software, or in the context menu of the Current Directory browser. For more information, see “Undoing the Checkout on UNIX Platforms”.

**Syntax**

undocheckout('filename')
undocheckout({'filename1','filename2', ...,'filenamen'})

**Description**

undocheckout('filename') makes the file filename available for checkout, where filename does not reflect any of the changes you made after you last checked it out. Use the full path for filename and include the file extension.

undocheckout({'filename1','filename2', ...,'filenamen'}) makes filename1 through filenamen available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full paths for the file names and include the file extensions.

**Examples**

Typing

undocheckout({'/myserver/mymfiles/clock.m', ...
 '/myserver/mymfiles/calendar.m'})

undoes the checkouts of /myserver/mymfiles/clock.m and /myserver/mymfiles/calendar.m from the source control system.

**See Also**

checkin, checkout

For Microsoft Windows platforms, use verctrl.
Convert Unicode characters to numeric bytes

`bytes = unicode2native(unicodestr)`

 takes a char vector of Unicode characters, `unicodestr`, converts it to the MATLAB default character encoding scheme, and returns the bytes as a `uint8` vector, `bytes`. Output vector `bytes` has the same general array shape as the `unicodestr` input. You can save the output of `unicode2native` to a file using the `fwrite` function.

`bytes = unicode2native(unicodestr, encoding)` converts the Unicode characters to the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string ('') or a name or alias for an encoding scheme. Some examples are 'UTF-8', 'latin1', 'US-ASCII', and 'Shift_JIS'. For common names and aliases, see the Web site http://www.iana.org/assignments/character-sets. If `encoding` is unspecified or is the empty string (''), the MATLAB default encoding scheme is used.

This example begins with two strings containing Unicode characters. It assumes that string `str1` contains text in a Western European language and string `str2` contains Japanese text. The example writes both strings into the same file, using the ISO-8859-1 character encoding scheme for the first string and the Shift-JIS encoding scheme for the second string. The example uses `unicode2native` to convert the two strings to the appropriate encoding schemes.

```matlab
fid = fopen('mixed.txt', 'w');
bytes1 = unicode2native(str1, 'ISO-8859-1');
fwrite(fid, bytes1, 'uint8');
bytes2 = unicode2native(str2, 'Shift_JIS');
fwrite(fid, bytes2, 'uint8');
fclose(fid);
```

See Also

`native2unicode`
union

Purpose
Find set union of two vectors

Syntax
\[ c = \text{union}(A, B) \]
\[ c = \text{union}(A, B, 'rows') \]
\[ [c, \text{ia}, \text{ib}] = \text{union}(\ldots) \]

Description
\( c = \text{union}(A, B) \) returns the combined values from \( A \) and \( B \) but with no repetitions. In set theoretic terms, \( c = A \cup B \). Inputs \( A \) and \( B \) can be numeric or character vectors or cell arrays of strings. The resulting vector is sorted in ascending order.

\( c = \text{union}(A, B, 'rows') \) when \( A \) and \( B \) are matrices with the same number of columns returns the combined rows from \( A \) and \( B \) with no repetitions. MATLAB ignores the rows flag for all cell arrays.

\[ [c, \text{ia}, \text{ib}] = \text{union}(\ldots) \] also returns index vectors \( \text{ia} \) and \( \text{ib} \) such that \( c = a(\text{ia}) \cup b(\text{ib}) \), or for row combinations, \( c = a(\text{ia,:}) \cup b(\text{ib,:}) \). If a value appears in both \( a \) and \( b \), \( \text{union} \) indexes its occurrence in \( b \). If a value appears more than once in \( b \) or in \( a \) (but not in \( b \)), \( \text{union} \) indexes the last occurrence of the value.

Remarks
Because NaN is considered to be not equal to itself, every occurrence of NaN in \( A \) or \( B \) is also included in the result \( c \).

Examples
\[ a = [-1 0 2 4 6]; \]
\[ b = [-1 0 1 3]; \]
\[ [c, \text{ia}, \text{ib}] = \text{union}(a, b); \]
\[ c = \]
\[-1 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 6 \]
\[ \text{ia} = \]
\[ 3 \quad 4 \quad 5 \]
\[ \text{ib} = \]
See Also

intersect, setdiff, setxor, unique, ismember, issorted
**Purpose**

Find unique elements of vector

**Syntax**

- `b = unique(A)`
- `b = unique(A, 'rows')`
- `[b, m, n] = unique(...)`
- `[b, m, n] = unique(..., occurrence)`

**Description**

`b = unique(A)` returns the same values as in A but with no repetitions. A can be a numeric or character array or a cell array of strings. If A is a vector or an array, b is a vector of unique values from A. If A is a cell array of strings, b is a cell vector of unique strings from A. The resulting vector b is sorted in ascending order and its elements are of the same class as A.

- `b = unique(A, 'rows')` returns the unique rows of A.
- `[b, m, n] = unique(...)` also returns index vectors m and n such that `b = A(m)` and `A = b(n)`. Each element of m is the greatest subscript such that `b = A(m)`. For row combinations, `b = A(m,:)` and `A = b(n,:)`.
- `[b, m, n] = unique(..., occurrence), where occurrence can be

  - 'first', which returns the vector m to index the first occurrence of each unique value in A, or
  - 'last', which returns the vector m to index the last occurrence.

If you do not specify occurrence, it defaults to 'last'.

You can specify 'rows' in the same command as 'first' or 'last'. The order of appearance in the argument list is not important.

**Examples**

```plaintext
A = [1 1 5 6 2 3 3 9 8 6 2 4]
A =
    1   1   5   6   2   3   3   9   8   6   2   4
```
Get a sorted vector of unique elements of A. Also get indices of the first elements in A that make up vector b, and the first elements in b that make up vector A:

\[
[b1, m1, n1] = \text{unique}(A, 'first')
\]

\[
b1 =
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 8 & 9
\end{bmatrix}
\]

\[
m1 =
\begin{bmatrix}
1 & 5 & 6 & 12 & 3 & 4 & 9 & 8
\end{bmatrix}
\]

\[
n1 =
\begin{bmatrix}
1 & 1 & 5 & 6 & 2 & 3 & 3 & 8 & 7 & 6 & 2 & 4
\end{bmatrix}
\]

Verify that \(b1 = A(m1)\) and \(A = b1(n1)\):

\[
\text{all}(b1 == A(m1)) && \text{all}(A == b1(n1))
\]

\[
ans =
1
\]

Get a sorted vector of unique elements of A. Also get indices of the last elements in A that make up vector b, and the last elements in b that make up vector A:

\[
[b2, m2, n2] = \text{unique}(A, 'last')
\]

\[
b2 =
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 8 & 9
\end{bmatrix}
\]

\[
m2 =
\begin{bmatrix}
2 & 11 & 7 & 12 & 3 & 10 & 9 & 8
\end{bmatrix}
\]

\[
n2 =
\begin{bmatrix}
1 & 1 & 5 & 6 & 2 & 3 & 3 & 8 & 7 & 6 & 2 & 4
\end{bmatrix}
\]

Verify that \(b2 = A(m2)\) and \(A = b2(n2)\):

\[
\text{all}(b2 == A(m2)) && \text{all}(A == b2(n2))
\]

\[
ans =
1
\]

Because NaNs are not equal to each other, \text{unique} treats them as unique elements.
unique([1 1 NaN NaN])
ans =
  1  NaN  NaN

See Also
intersect, ismember, sort, issorted, setdiff, setxor, union
**Purpose**

Execute UNIX command and return result

**Syntax**

```matlab
unix command
status = unix('command')
[status, result] = unix('command')
[status, result] = unix('command', '-echo')
```
unix

**Description**

`unix` command calls upon the UNIX\(^{28}\) operating system to execute the given command.

```matlab
status = unix('command')
```
returns completion status to the `status` variable.

```matlab
[status, result] = unix('command')
```
returns the standard output to the `result` variable, in addition to completion status.

```matlab
[status, result] = unix('command', '-echo')
```
displays the results in the Command Window as it executes, and assigns the results to `w`.

**Note** The MATLAB software uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.

**Examples**

List all users that are currently logged in.

```matlab
[s,w] = unix('who');
```
MATLAB returns 0 (success) in `s` and a string containing the list of users in `w`.

In this example

```matlab
[s,w] = unix('why')
```
```matlab
s =
    1
w =
why: Command not found.
```
MATLAB returns a nonzero value in `s` to indicate failure, and returns an error message in `w` because `why` is not a UNIX command.

\(^{28}\) UNIX is a registered trademark of The Open Group in the United States and other countries.
See Also
dos, ! (exclamation point), perl, system

“Running External Programs” in the MATLAB Desktop Tools and Development Environment documentation
unloadlibrary

**Purpose**
Unload shared library from memory

**Syntax**

```
unloadlibrary('libname')
unloadlibrary libname
```

**Description**

`unloadlibrary('libname')` unloads the shared library `libname` from memory. If you need to use functions in this library, you must reload the library using the `loadlibrary` function.

`unloadlibrary libname` is the command format for this function.

If you used an alias when initially loading the library, then you must use that alias for the `libname` argument.

**Examples**

Load the MATLAB sample shared library, `shrlibsample`. Call one of its functions, and then unload the library:

```
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibs SMP shrlibs SMP.h

s.p1 = 476; s.p2 = -299; s.p3 = 1000;
calllib('shrlibs SMP', 'addStructFields', s)
ans =
    1177

unloadlibrary shrlibs SMP
```

**See Also**

`loadlibrary`, `libisloaded`
Purpose: Convert edge matrix to coordinate and Laplacian matrices

Syntax: 

\[ [L, XY] = \text{UNMESH}(E) \]

Description: 

\([L, XY] = \text{UNMESH}(E)\) returns the Laplacian matrix \(L\) and mesh vertex coordinate matrix \(XY\) for the \(M\)-by-4 edge matrix \(E\). Each row of the edge matrix must contain the coordinates \([x_1 \ y_1 \ x_2 \ y_2]\) of the edge endpoints.

Inputs: 

\(E\) \hspace{1cm} \text{M-by-4 edge matrix } E.

Outputs: 

\(L\) \hspace{1cm} \text{Laplacian matrix representation of the graph.}

\(XY\) \hspace{1cm} \text{Mesh vertex coordinate matrix.}

Examples: 

Take a simple example of a square with vertices at (1,1), (1,−1), (−1,−1), and (−1,1), where the connections between vertices are the four perpendicular edges of the square plus one diagonal connection between (−1, −1) and (1,1).
The edge matrix $E$ for this graph is:

$$E=\begin{bmatrix} 1 & 1 & 1 & -1; & \% \text{ edge from 1 to 2} \\
1 & -1 & -1 & -1; & \% \text{ edge from 2 to 3} \\
-1 & -1 & -1 & 1; & \% \text{ edge from 3 to 4} \\
-1 & -1 & 1 & 1; & \% \text{ edge from 4 to 1} \\
-1 & 1 & 1 & 1; & \% \text{ edge from 3 to 1} \end{bmatrix}$$

Use `unmesh` to create the output matrices,

$$[A,XY]=\text{unmesh}(E);$$

4 vertices:

4/4

The Laplacian matrix is defined as
\[
L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}
\]

\text{unmesh} returns the Laplacian matrix \( L \) in sparse notation.

\[
L = \begin{pmatrix}
(1,1) & 3 \\
(2,1) & -1 \\
(3,1) & -1 \\
(4,1) & -1 \\
(1,2) & -1 \\
(2,2) & 2 \\
(4,2) & -1 \\
(1,3) & -1 \\
(3,3) & 2 \\
(4,3) & -1 \\
(1,4) & -1 \\
(2,4) & -1 \\
(3,4) & -1 
\end{pmatrix}
\]

To see \( L \) in regular matrix notation, use the \text{full} command.

\[
\text{full}(L)
\]

\[
\text{ans} = \begin{pmatrix}
3 & -1 & -1 & -1 \\
-1 & 2 & 0 & -1 \\
-1 & 0 & 2 & -1 \\
-1 & -1 & -1 & 3 
\end{pmatrix}
\]

The mesh coordinate matrix \( XY \) returns the coordinates of the corners of the square.
unmesh

XY

XY =

-1 -1
-1  1
 1 -1
 1  1

See Also

gplot
treeplot
Purpose
Piecewise polynomial details

Syntax
\[ \text{[breaks,coefs,l,k,d]} = \text{unmkpp(pp)} \]

Description
\[ \text{[breaks,coefs,l,k,d]} = \text{unmkpp(pp)} \]
extracts, from the piecewise polynomial \( pp \), its breaks \( \text{breaks} \), coefficients \( \text{coefs} \), number of pieces \( l \), order \( k \), and dimension \( d \) of its target. Create \( pp \) using \text{spline} or the spline utility \text{mkpp}.

Examples
This example creates a description of the quadratic polynomial
\[
\frac{-x^2}{4} + x
\]
as a piecewise polynomial \( pp \), then extracts the details of that description.

\[
\text{pp} = \text{mkpp([-8 -4],[-1/4 1 0])};
\]
\[
\text{[breaks,coefs,l,k,d]} = \text{unmkpp(pp)}
\]

\[
\text{breaks} =
\]
\[-8\quad -4\]

\[
\text{coefs} =
\]
\[-0.2500\quad 1.0000\quad 0\]

\[
\text{l} = 1
\]

\[
\text{k} = 3
\]

\[
\text{d} = 1
\]

See Also
\text{mkpp, ppval, spline}
**unregisterallevents**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Unregister all event handlers for COM object event at run-time</th>
</tr>
</thead>
</table>
| Syntax  | h.unregisterallevents  
unregisterallevents(h) |
| Description | h.unregisterallevents unregisters all events that have previously been registered with COM object, h. After calling unregisterallevents, the object no longer responds to any events until you register them again using the registerevent function. unregisterallevents(h) is an alternate syntax. |
| Remarks  | COM functions are available on Microsoft Windows systems only. |
| Examples | **mwsamp Control Example**  
Create an mwsamp control, registering three events and their respective handler routines. Use the eventlisteners function to see the event handler used by each event:  

```matlab  
f = figure ('position', [100 200 200 200]);  
h = actxcontrol('mwsamp.mwsampctrl.2', ...  
[0 0 200 200], f, ...  
{'Click' 'myclick'; 'Db1Click' 'my2click'; ...  
'MouseDown' 'mymoused'});  
h.eventlisteners  
```

MATLAB displays:

```matlab  
ans =  
'click'       'myclick'  
'dblclick'     'my2click'  
'mousedown'   'mymoused'  
```

Unregister all of these events at once with unregisterallevents. Now, calling eventlisteners returns an empty cell array, indicating that there are no longer any events registered with the control:
h.unregisterallevents;
h.eventlisteners

MATLAB displays:

ans =
{
}

To unregister specific events, use the unregisterevent function.

**Workbook Events Example**

Create a Microsoft Excel Workbook object and register some events.

```matlab
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = wbs.Add;
w.b.registerevent({'Activate' 'EvtActivateHndlr'; ...
                  'Deactivate' 'EvtDeactivateHndlr'})
w.b.eventlisteners
```

MATLAB shows the events registered to their corresponding event handlers.

```matlab
ans =

'Activate'    'EvtActivateHndlr'
'Deactivate'  'EvtDeactivateHndlr'
```

Use unregisterallevents to clear the events.

```matlab
wb.unregisterallevents
wb.eventlisteners
```

MATLAB displays an empty cell array, showing that no events are registered.

```matlab
ans =
```
unregisterallevents

{}

See Also  

events (COM), eventlisteners, registerevent, unregisterevent, isevent
**Purpose**
Unregister event handler for COM object event at run-time

**Syntax**

```
unregistrerevent(event_handler)
unregistrerevent(h, event_handler)
```

**Description**

`h.unregistrerevent(event_handler)` unregisters certain event handler routines with their corresponding events. Once you unregister an event, the object no longer responds to any further occurrences of the event.

`unregistrerevent(h, event_handler)` is an alternate syntax.

You can unregister events at any time after a control has been created. The `event_handler` argument, which is a cell array, specifies both events and event handlers. For example:

```
    h.unregistrerevent({'event_name', @event_handler});
```

See “Writing Event Handlers” in the External Interfaces documentation.

You must specify events in the `event_handler` argument using the names of the events. Strings used in the `event_handler` argument are not case sensitive. Unlike `actxcontrol` and `registrerevent`, `unregistrerevent` does not accept numeric event identifiers.

**Remarks**

COM functions are available on Microsoft Windows systems only.

**Examples**

**Control Example**

Create an `mwsamp` control and register all events with the same handler routine, `sampev`. Use `eventlisteners` to see the event handler used by each event. In this case, each event, when fired, calls `sampev.m`:

```
    f = figure ('position', [100 200 200 200]);
    h = actxcontrol('mwsamp.mwsampctrl.2', ...
                    [0 0 200 200], f, ...
                    'sampev');
    h.eventlisteners
```
MATLAB displays:

```
ans =
    'Click'    'sampev'
    'Db1Click' 'sampev'
    'MouseDown' 'sampev'
    'Event_Args' 'sampev'
```

Unregister just the `dblclick` event. Now, when you list the registered events using `eventlisteners`, `dblclick` is no longer registered and the control does not respond when you double-click the mouse over it:

```matlab
h.unregisterevent({'dblclick' 'sampev'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'Click'    'sampev'
    'MouseDown' 'sampev'
    'Event_Args' 'sampev'
```

This time, register the `click` and `dblclick` events with a different event handler for `myclick` and `my2click`, respectively:

```matlab
h.unregisterallevents;
h.registerevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'click'    'myclick'
    'dblclick' 'my2click'
```

You can unregister these same events by specifying event names and their handler routines in a cell array. `eventlisteners` now returns
an empty cell array, meaning no events are registered for the mwsamp control:

    h.unregisterevent({'click' 'myclick'; ...
                    'dblclick' 'my2click'});
    h.eventlisteners

MATLAB displays:

    ans = 
    {} 

**Workbook Events Example**

Create a Microsoft Excel Workbook object:

    myApp = actxserver('Excel.Application');
    wbs = myApp.Workbooks;
    wb = wbs.Add;

Register two events with the your event handler routines, EvtActivateHndlr and EvtDeactivateHndlr.

    wb.registerevent({'Activate' 'EvtActivateHndlr'; ...
                      'Deactivate' 'EvtDeactivateHndlr'})
    wb.eventlisteners

MATLAB shows the events with the corresponding event handlers.

    ans =

    'Activate'    'EvtActivateHndlr'
    'Deactivate'  'EvtDeactivateHndlr'

Next, unregister the Deactivate event handler.

    wb.unregisterevent({'Deactivate' 'EvtDeactivateHndlr'})
    wb.eventlisteners
MATLAB shows the remaining registered event (Activate) with its corresponding event handler.

ans =

    'Activate'    'EvtActivateHndlr'

See Also

events (COM), eventlisteners, registerevent, unregisterallevents, isevent
**Purpose**

Extract contents of tar file

**Syntax**

```
untar(tarfilename)
untar(tarfilename, outputdir)
untar(url, ...)
filenames = untar(...)
```

**Description**

`untar(tarfilename)` extracts the archived contents of `tarfilename` into the current directory and sets the files’ attributes. It overwrites any existing files with the same names as those in the archive if the existing files’ attributes and ownerships permit it. For example, if you rerun `untar` on the same `tarfilename`, MATLAB software does not overwrite files with a read-only attribute; instead, `untar` displays a warning for such files. On Microsoft Windows platforms, the hidden, system, and archive attributes are not set.

`targilename` is a string specifying the name of the tar file. `tarfilename` is gunzipped to a temporary directory and deleted if its extension ends in `.tgz` or `.gz`. If an extension is omitted, `untar` searches for `tarfilename` appended with `.tgz`, `.tar.gz`, or `.tar`. `tarfilename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

`untar(tarfilename, outputdir)` uncompresses the archive `tarfilename` into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`untar(url, ...)` extracts the tar archive from an Internet URL. The URL must include the protocol type (for example, `'http://'` or `'ftp://'`). MATLAB downloads the URL is to a temporary directory, and then deletes it.

`filenames = untar(...)` extracts the tar archive and returns the names of the extracted files in the string cell array `filenames`. If `outputdir` specifies a relative path, `filenames` contains the relative path. If `outputdir` specifies an absolute path, `filenames` contains the absolute path.
Examples

Using tar and untar to Copy Files

Copy all .m files in the current directory to the directory backup.

```
tar('mymfiles.tar.gz','*.m');
untar('mymfiles','backup');
```

Using untar with URL

Run untar to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory ncm.

```
url = 'http://www.mathworks.com/moler/ncm.tar.gz';
ncmFiles = untar(url,'ncm')
```

See Also
gzip, gunzip, tar, unzip, zip
**Purpose**  
Correct phase angles to produce smoother phase plots

**Syntax**  

Q = unwrap(P)  
Q = unwrap(P,tol)  
Q = unwrap(P,[],dim)  
Q = unwrap(P,tol,dim)

**Description**  

Q = unwrap(P) corrects the radian phase angles in a vector P by adding multiples of $\pm 2\pi$ when absolute jumps between consecutive elements of P are greater than or equal to the default jump tolerance of $\pi$ radians. If P is a matrix, unwrap operates columnwise. If P is a multidimensional array, unwrap operates on the first nonsingleton dimension.

Q = unwrap(P,tol) uses a jump tolerance tol instead of the default value, $\pi$.

Q = unwrap(P,[],dim) unwraps along dim using the default tolerance.

Q = unwrap(P,tol,dim) uses a jump tolerance of tol.

**Note**  
A jump tolerance less than $\pi$ has the same effect as a tolerance of $\pi$. For a tolerance less than $\pi$, if a jump is greater than the tolerance but less than $\pi$, adding $\pm 2\pi$ would result in a jump larger than the existing one, so unwrap chooses the current point. If you want to eliminate jumps that are less than $\pi$, try using a finer grid in the domain.

**Examples**  

**Example 1**

The following phase data comes from the frequency response of a third-order transfer function. The phase curve jumps $3.5873$ radians between $w = 3.0$ and $w = 3.5$, from $-1.8621$ to $1.7252$.

```matlab
w = [0:.2:3,3.5:1:10];
p = [0
    -1.5728
    -1.5747
    -1.5772
```
unwrap

-1.5790
-1.5816
-1.5852
-1.5877
-1.5922
-1.5976
-1.6044
-1.6129
-1.6269
-1.6512
-1.6998
-1.8621
1.7252
1.6124
1.5930
1.5916
1.5708
1.5708
1.5708 ];
semilogx(w,p,'b*-'), hold
Using `unwrap` to correct the phase angle, the resulting jump is 2.6959, which is less than the default jump tolerance \(\pi\). This figure plots the new curve over the original curve.

\[
\text{semilogx}(w, \text{unwrap}(p), 'r*-')
\]
Note If you have the “Control System Toolbox”, you can create the data for this example with the following code.

```
    h = freqresp(tf(1,[1 .1 10 0]));
    p = angle(h(:));
```

---

**Example 2**

Array $P$ features smoothly increasing phase angles except for discontinuities at elements $(3,1)$ and $(1,2)$.

```
    P = [ 0 7.0686 1.5708 2.3562
          0.1963 0.9817 1.7671 2.5525
         6.6759 1.1781 1.9635 2.7489
        0.5890 1.3744 2.1598 2.9452 ]
```
The function $Q = \text{unwrap}(P)$ eliminates these discontinuities.

$$Q =
\begin{array}{cccc}
0 & 7.0686 & 1.5708 & 2.3562 \\
0.1963 & 7.2649 & 1.7671 & 2.5525 \\
0.3927 & 7.4613 & 1.9635 & 2.7489 \\
0.5890 & 7.6576 & 2.1598 & 2.9452 \\
\end{array}
$$

See Also

abs, angle
**unzip**

**Purpose**
Extract contents of zip file

**Syntax**

unzip(zipfilename)
unzip(zipfilename, outputdir)
unzip(url, ...)
filenames = unzip(...)

**Description**

unzip(zipfilename) extracts the archived contents of zipfilename into the current directory and sets the files’ attributes, preserving the timestamps. It overwrites any existing files with the same names as those in the archive if the existing files’ attributes and ownerships permit it. For example, files from rerunning unzip on the same zip filename do not overwrite any of those files that have a read-only attribute; instead, unzip issues a warning for such files.

zipfilename is a string specifying the name of the zip file. The .zip extension is appended to zipfilename if omitted. zipfilename can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

unzip(zipfilename, outputdir) extracts the contents of zipfilename into the directory outputdir.

unzip(url, ...) extracts the-zipped contents from an Internet URL. The URL must include the protocol type (for example, http://). The URL is downloaded to the temp directory and deleted.

filenames = unzip(...) extracts the zip archive and returns the names of the extracted files in the string cell array filenames. If outputdir specifies a relative path, filenames contains the relative path. If outputdir specifies an absolute path, filenames contains the absolute path.

unzip does not support password-protected or encrypted zip archives.

**Examples**

**Using zip and unzip to Copy Files**

Copy the demos HTML files to the directory archive:

```
% Zip the demos html files to demos.zip
```
zip('demos.zip','*.html',fullfile(matlabroot,'demos'))
% Unzip demos.zip to the 'directory' archive
unzip('demos','archive')

Using unzip with URL

Run unzip to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory ncm.

url = 'http://www.mathworks.com/moler/ncm.zip';
ncmFiles = unzip(url,'ncm')

See Also

fileattrib, gzip, gunzip, tar, untar, zip
**Purpose**
Convert string to uppercase

**Syntax**
\[
t = \text{upper}('\text{str}') \\
B = \text{upper}(A)
\]

**Description**
\[
t = \text{upper}('\text{str}')\] converts any lowercase characters in the string \text{str} to the corresponding uppercase characters and leaves all other characters unchanged.

\[
B = \text{upper}(A)\] when \text{A} is a cell array of strings, returns a cell array the same size as \text{A} containing the result of applying \text{upper} to each string within \text{A}.

**Examples**
\[
\text{upper}('attention!')\] is ATTENTION!.

**Remarks**
Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

**See Also**
\[
\text{lower}
\]
Purpose
Read content at URL

Syntax
\[
s = \text{urlread}'url'\]
\[
s = \text{urlread}'url','method','params'\]
\[
[s,\text{status}] = \text{urlread}(...)\]

Description
\(s = \text{urlread}'url'\) reads the content at a URL into the string \(s\). If the server returns binary data, \(s\) will be unreadable.

\(s = \text{urlread}'url','method','params'\) reads the content at a URL into the string \(s\), passing information to the server as part of the request where \(method\) can be get or post, and \(params\) is a cell array of parameter name/parameter value pairs.

\([s,\text{status}] = \text{urlread}(...)\) catches any errors and returns the error code.

Note
If you need to specify a proxy server to connect to the Internet, select File -> Preferences -> Web and enter your proxy information. Use this feature if you have a firewall.

Examples

Download Content from Web Page
Use urlread to download the contents of the Authors list at the MATLAB Central File Exchange:

\[
\text{urlstring} = \text{sprintf}'\%s\%s',...\]
\[
'\text{http://www.mathworks.com/matlabcentral/}',...\]
\[
'\text{fileexchange/authors}')\;
\]
\[
s = \text{urlread(urlstring)};\]

Download Content from File on FTP Server

\[
\text{page} = \text{'ftp://ftp.mathworks.com/pub/tech-support/docexamples/README}\]
\[
s=\text{urlread(page)};\]
urlread

MATLAB displays:

The MathWorks FTP site has recently changed. If you are trying to access one of the following pages, please see the corresponding new link:

====================
OLD:
...

File updated: 15 November 2004

Download Content from Local File

s = urlread('file:///c:/winnt/matlab.ini')

See Also

urlwrite, web
tcpip if the Instrument Control Toolbox™ is installed
urlwrite

**Purpose**  
Save contents of URL to file

**Syntax**  
urlwrite('url','filename')

\[f = \text{urlwrite('url','filename')}\]

\[f = \text{urlwrite('url','method','params')}\]

\[[f,\text{status}] = \text{urlwrite(...)}\]

**Description**  
urlwrite('url','filename') reads the contents of the specified URL, saving the contents to filename. If you do not specify the path for filename, the file is saved in the MATLAB current directory.

\[f = \text{urlwrite('url','filename')}\] reads the contents of the specified URL, saving the contents to filename and assigning filename to f.

\[f = \text{urlwrite('url','method','params')}\] saves the contents of the specified URL to filename, passing information to the server as part of the request where method can be get or post, and params is a cell array of parameter name/parameter value pairs.

\[[f,\text{status}] = \text{urlwrite(...)}\] catches any errors and returns the error code.

---

**Note**  
If you need to specify a proxy server to connect to the Internet, select File -> Preferences -> Web and enter your proxy information. Use this feature if you have a firewall.

**Examples**  
Download the files submitted to the MATLAB Central File Exchange, saving the results to samples.html in the MATLAB current directory.

\[\text{urlstring} = \text{sprintf('%s%s', ..., 'http://www.mathworks.com/matlabcentral/', ... 'fileexchange/Category.jsp?type=category&id=1');}\]

\[\text{urlwrite(urlstring, 'samples.html');}\]

View the file in the Help browser.
urlwrite

```
open('samples.html')
```

**See Also**

urlread
**Purpose**
Determine whether Sun Java feature is supported in MATLAB software

**Syntax**
```
usejava(feature)
```

**Description**
```
usejava(feature) returns 1 if the specified feature is supported and 0 otherwise. Possible feature arguments are shown in the following table.
```

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'awt'</td>
<td>Abstract Window Toolkit components(^1) are available</td>
</tr>
<tr>
<td>'desktop'</td>
<td>The MATLAB interactive desktop is running</td>
</tr>
<tr>
<td>'jvm'</td>
<td>The Java Virtual Machine software (JVM) is running</td>
</tr>
<tr>
<td>'swing'</td>
<td>Swing components(^2) are available</td>
</tr>
</tbody>
</table>

1. Java GUI components in the Abstract Window Toolkit
2. Java lightweight GUI components in the Java Foundation Classes

**Examples**
The following conditional code ensures that the AWT's GUI components are available before the M-file attempts to display a Java Frame.

```
if usejava('awt')
    myFrame = java.awt.Frame;
else
    disp('Unable to open a Java Frame');
end
```

The next example is part of an M-file that includes Java code. It fails gracefully when run in a MATLAB session that does not have access to JVM software.

```
if ~usejava('jvm')
    error([mfilename ' requires Java to run.']);
end
```
See Also

javachk
Purpose  View or change user portion of search path

Syntax  
userpath
userpath('newpath')
userpath('reset')
userpath('clear')

Description  userpath returns a string specifying the user portion of the search path. The user portion of the search path is the first directory on the search path, above the directories supplied by The MathWorks. The default directory is My Documents/MATLAB on Microsoft Windows platforms, and Documents/MATLAB on Microsoft Windows Vista™ platforms.
On Apple Macintosh and UNIX platforms, the default value is `userhome/Documents/MATLAB`. If you remove the userpath directory from the search path and save the changes to the path, it also clears the value of userpath. You can define the userpath directory to also be the MATLAB startup directory. On Windows platforms, userpath is the startup directory, unless the startup directory is otherwise specified, such as by the MATLAB shortcut properties Start in field. On UNIX and Macintosh platforms, the startup directory is userpath if the value of the environment variable `MATLAB_USE_USERPATH` is set to 1 prior to startup and if the startup directory is not otherwise specified, such as via a startup.m file. On Macintosh and UNIX platforms, you can automatically add additional subdirectories to the top of the search path upon startup by specifying the path for the subdirectories via the `MATLABPATH` environment variable.

`userpath('newpath')` sets the userpath value to newpath. The newpath directory appears at the top of the search path immediately and at startup in future sessions, and MATLAB removes the directory previously specified by userpath from the search path. newpath cannot be a relative path. `userpath('newpath')` does not work when the `-nojvm` startup option is used. Upon the next startup, newpath, can become the current directory, as described in the syntax for userpath with no arguments.

`userpath('reset')` sets the userpath value to the default for that platform, creating the `Documents/MATLAB` (or `My Documents/MATLAB`) directory if it does not exist. MATLAB immediately adds the default directory to the top of the search path, and also adds it to the path at startup in future sessions; it can become the startup directory as described for the userpath syntax with no arguments. MATLAB removes the directory previously specified by userpath from the search path. `userpath('reset')` does not work when the `-nojvm` startup option is used.

`userpath('clear')` clears the value for userpath. MATLAB removes the directory previously specified by userpath from the search path.

29. UNIX is a registered trademark of The Open Group in the United States and other countries.
This does not work when the -nojvm startup option is used. You can otherwise specify the startup directory—see “Startup Directory for the MATLAB Program”.

**Examples**

- “Viewing userpath” on page 2-4063
- “Setting a New Value for userpath” on page 2-4064
- “Clearing the Value for userpath, and Specifying a New Startup Directory on Windows Platforms” on page 2-4065
- “Removing userpath from the Search Path; Resets the Startup Directory” on page 2-4066
- “Assigning userpath as the Startup Directory on a UNIX or Macintosh Platform” on page 2-4068
- “Adding Directories to the Search Path Upon Startup on a UNIX or Macintosh Platform” on page 2-4069

**Viewing userpath**

This example assumes userpath is set to the default value on the Windows XP platform, My Documents\MATLAB. Start MATLAB and run

```
cd
```

MATLAB displays the current directory

```
H:\My Documents\MATLAB
```

where H is the drive at which My Documents is located for this example. This is the directory specified by userpath. To confirm, run

```
userpath
```

and MATLAB returns

```
H:\My Documents\MATLAB;
```

Run
and MATLAB displays the search path; the `userpath` portion is at the top:

```
MATLABPATH

H:\My Documents\MATLAB
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
```

### Setting a New Value for `userpath`

This example assumes `userpath` is set to the default value on the Windows XP platform, `My Documents\MATLAB`. To change the value from the default for `userpath` to `C:\Research_Project`, run

```
userpath('C:\Research_Project')
```

To view the effect of the change on the search path, run

```
path
```

and MATLAB displays the search path, with the new value for `userpath` portion at the top:

```
MATLABPATH

C:\Research_Project
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Note that MATLAB automatically removed the previous value of `userpath`, `H:\My Documents\MATLAB`, from the search path when you assigned a new value to `userpath`. The next time you start MATLAB, the current directory will be `C:\Research_Project` on Windows platforms.
Clearing the Value for userpath, and Specifying a New Startup Directory on Windows Platforms

Assume userpath is set to the default value and you do not want any directories to be added to the search path upon startup. To confirm the default is currently set, run

    userpath

and MATLAB returns

    H:\My Documents\MATLAB

Note the userpath directory at the top of the search path by running

    path

MATLAB returns

    MATLABPATH

    H:\My Documents\MATLAB
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
    C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
    ...

To clear the value, run

    userpath('clear')

To verify the result, run

    userpath

MATLAB returns

    ans = 
    ''

Confirm the userpath directory was removed from the path by running:
MATLAB returns

MATLABPATH

C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...

After clearing the userpath value, unless you otherwise specify the startup directory, the startup directory will be the desktop on Windows platforms. There are a number of ways to specify the startup directory. For example, right-click the Windows shortcut icon for MATLAB and select Properties from the context menu. In the Properties dialog box Shortcut tab, enter the full path to the new startup directory in the Start in field, for example, I:\my_matlab_files\my_mfiles. The next time you start MATLAB, the current directory will be I:\my_matlab_files\my_mfiles, but that directory will not be on the search path. Note that you do not have to clear the userpath to specify a different startup directory; when you otherwise specify a startup directory, the userpath directory is added to the search path upon startup, but is not the startup directory.

Removing userpath from the Search Path; Resets the Startup Directory

In this example, assume userpath is set to the default value and you remove the userpath directory from the search path, then save the changes. This has the same effect as clearing the value for userpath. To confirm the default is currently set, run

userpath

and MATLAB returns

H:\My Documents\MATLAB

Note the userpath directory at the top of the search path by running
path

MATLAB returns

MATLABPATH

H:\My Documents\MATLAB
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...

Remove H:\My Documents\MATLAB from the search path and confirm the result by running

rmpath('H:\My Documents\MATLAB')
path

MATLAB returns

MATLABPATH

C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...

Running

userpath

at this point shows the value is still set

H:\My Documents\MATLAB

Save changes to the path by running

savepath

Now when you run

userpath
MATLAB returns

    ans =
    ``

showing the value is now cleared. Removing the directory from the search path and saving the changes to the path has the same effect as clearing the value for userpath. At the next startup, the startup directory will *not* be H:\My Documents\MATLAB, and H:\My Documents\MATLAB will *not* be on the search path.

**Assigning userpath as the Startup Directory on a UNIX or Macintosh Platform**

This example assumes userpath is set to the default value on a Macintosh platform and that you start MATLAB using a bash X11 shell, where smith is the home directory. Set the MATLAB_USE_USERPATH environment variable so that userpath will be used as the startup directory:

    export MATLAB_USE_USERPATH=1

From that shell, start MATLAB. After MATLAB starts, verify its current directory by running

    pwd

MATLAB returns

    /Users/smith/Documents/MATLAB

That is the value defined for userpath, which you can confirm by running

    userpath

MATLAB returns

    /Users/smith/Documents/MATLAB
The userpath is at the top of the search path, which you can confirm by running

```matlab
path
```

MATLAB returns

```
/Users/smith/Documents/MATLAB
/Users/smith/Applications/MATLAB/R2009a/toolbox/matlab/general
/Users/smith/Applications/MATLAB/R2009a/toolbox/matlab/ops
...
```

**Adding Directories to the Search Path Upon Startup on a UNIX or Macintosh Platform**

This example assumes userpath is set to the default value on a UNIX platform with a csh shell, where j is the user’s home directory.

To add additional directories to the search path upon startup, for example, `/home/j/Documents/MATLAB/mine` and `/home/j/Documents/MATLAB/mine/research`, run the following in an X11 terminal:

```matlab
setenv MATLABPATH \/home/j/Documents/MATLAB/mine:\/home/j/Documents/MATLAB/mine/research
```

Separate multiple directories using a : (colon).

MATLAB displays

```matlab
MATLABPATH
```

```
/home/j/Documents/MATLAB
/home/j/Documents/MATLAB/mine
/home/j/Documents/MATLAB/mine/research
/home/j/Applications/MATLAB/R2009a/toolbox/matlab/general
/home/j/Applications/MATLAB/R2009a/toolbox/matlab/ops
...
```

**See Also**

addpath, path, pathtool, rmpath, savepath, startup,
“Startup and Shutdown” and “Search Path” in the MATLAB Desktop Tools and Development Environment documentation
validateattributes

**Purpose**
Check validity of array

**Syntax**

validateattributes(A, classes, attributes)
validateattributes(A, classes, attributes, position)
validateattributes(A, classes, attributes, funname)
validateattributes(A, classes, attributes, funname, varname)
validateattributes(A, classes, attributes, funname, varname, position)

**Description**

validateattributes(A, classes, attributes) validates that array A belongs to at least one of the classes specified by the classes input and has all of the attributes specified by the attributes input. If the validation succeeds, the command completes without displaying any output and without throwing an error. If the validation does not succeed, MATLAB issues a formatted error message.

The classes input is a cell array of one or more strings, each string containing the name of a MATLAB class (i.e., one of the 15 MATLAB data types), the name of a MATLAB class, or the keyword numeric. (See the table Class Values on page 2-4072.)

The attributes input is a cell array of one or more strings, each string describing an array attribute. Size validation requires two inputs: the 'size' keyword and the length of each dimension specified within square brackets (e.g., {'size', [4,3,7]}). Value range validation requires two inputs for each aspect of the range being validated (e.g., {'>', 10, '<=', 65}). (See the table Attribute Values on page 2-4073.)

validateattributes(A, classes, attributes, position) validates array A and, if the validation fails, displays an error message that includes the position of the failing variable in the function argument list. The position input must be a positive integer.

validateattributes(A, classes, attributes, funname) validates array A and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname). The funname input must be a string enclosed in single quotation marks.
validateattributes(A, classes, attributes, funname, varname) validates array A and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname), and the name of the variable being validated (varname). The funname and varname inputs must be strings enclosed in single quotation marks.

validateattributes(A, classes, attributes, funname, varname, position) validates array A and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname), the name of the variable being validated (varname), and the position of this variable in the function argument list (position). The funname and varname inputs must be strings enclosed in single quotation marks. The position input must be a positive integer.

**Class Values**

<table>
<thead>
<tr>
<th>classes Argument</th>
<th>Contents of Array A</th>
</tr>
</thead>
<tbody>
<tr>
<td>'numeric'</td>
<td>Any numeric value</td>
</tr>
<tr>
<td>'single'</td>
<td>Single-precision number</td>
</tr>
<tr>
<td>'double'</td>
<td>Double-precision number</td>
</tr>
<tr>
<td>'int8'</td>
<td>Signed 8-bit integer</td>
</tr>
<tr>
<td>'int16'</td>
<td>Signed 16-bit integer</td>
</tr>
<tr>
<td>'int32'</td>
<td>Signed 32-bit integer</td>
</tr>
<tr>
<td>'int64'</td>
<td>Signed 64-bit integer</td>
</tr>
<tr>
<td>'uint8'</td>
<td>Unsigned 8-bit integer</td>
</tr>
<tr>
<td>'uint16'</td>
<td>Unsigned 16-bit integer</td>
</tr>
<tr>
<td>'uint32'</td>
<td>Unsigned 32-bit integer</td>
</tr>
<tr>
<td>'uint64'</td>
<td>Unsigned 64-bit integer</td>
</tr>
<tr>
<td>'logical'</td>
<td>Logical true or false</td>
</tr>
</tbody>
</table>
Class Values (Continued)

<table>
<thead>
<tr>
<th>classes Argument</th>
<th>Contents of Array A</th>
</tr>
</thead>
<tbody>
<tr>
<td>'char'</td>
<td>Character or string</td>
</tr>
<tr>
<td>'struct'</td>
<td>MATLAB structure</td>
</tr>
<tr>
<td>'cell'</td>
<td>Cell array</td>
</tr>
<tr>
<td>'function_handle'</td>
<td>Scalar function handle</td>
</tr>
<tr>
<td>class name</td>
<td>Object of any MATLAB class</td>
</tr>
</tbody>
</table>

Attribute Values

<table>
<thead>
<tr>
<th>attributes Argument</th>
<th>Description of array A</th>
</tr>
</thead>
<tbody>
<tr>
<td>'&gt;', N</td>
<td>Array in which all values are greater than N.</td>
</tr>
<tr>
<td>'&gt;=', N</td>
<td>Array in which all values are greater than or equal to N.</td>
</tr>
<tr>
<td>'&lt;', N</td>
<td>Array in which all values are less than N.</td>
</tr>
<tr>
<td>'&lt;=', N</td>
<td>Array in which all values are less than or equal to N.</td>
</tr>
<tr>
<td>'2d'</td>
<td>Array having dimensions M-by-N (includes scalars, vectors, 2-D matrices, and empty arrays)</td>
</tr>
<tr>
<td>'column'</td>
<td>Array having dimensions N-by-1</td>
</tr>
<tr>
<td>'even'</td>
<td>Numeric or logical array in which all elements are even (includes zero)</td>
</tr>
<tr>
<td>'finite'</td>
<td>Numeric array in which all elements are finite</td>
</tr>
<tr>
<td>'integer'</td>
<td>Numeric array in which all elements are integer-valued</td>
</tr>
<tr>
<td>'nonempty'</td>
<td>Array having no dimension equal to zero</td>
</tr>
<tr>
<td>'nonnan'</td>
<td>Numeric array in which there are no elements equal to NaN (Not a Number)</td>
</tr>
</tbody>
</table>
### Attribute Values (Continued)

<table>
<thead>
<tr>
<th>attributes Argument</th>
<th>Description of array A</th>
</tr>
</thead>
<tbody>
<tr>
<td>'nonnegative'</td>
<td>Numeric array in which all elements are zero or greater than zero</td>
</tr>
<tr>
<td>'nonsparse'</td>
<td>Array that is not sparse</td>
</tr>
<tr>
<td>'nonzero'</td>
<td>Numeric or logical array in which all elements are less than or greater than zero</td>
</tr>
<tr>
<td>'odd'</td>
<td>Numeric or logical array in which all elements are odd integers</td>
</tr>
<tr>
<td>'positive'</td>
<td>Numeric or logical array in which all elements are greater than zero</td>
</tr>
<tr>
<td>'real'</td>
<td>Numeric array in which all elements are real</td>
</tr>
<tr>
<td>'row'</td>
<td>Array having dimensions 1-by-N</td>
</tr>
<tr>
<td>'scalar'</td>
<td>Array having dimensions 1-by-1</td>
</tr>
<tr>
<td>'size', [M,N,...]</td>
<td>Array having dimensions M-by-N-by-.....</td>
</tr>
<tr>
<td>'vector'</td>
<td>Array having dimensions N-by-1 or 1-by-N (includes scalar arrays)</td>
</tr>
</tbody>
</table>

Numeric properties, such as positive and nonnan, do not apply to strings. If you attempt to validate numeric properties on a string, validateattributes generates an error.

### Examples

#### Example 1

In this example, the empl_profile1 function compares the values passed in each argument to the specified classes and attributes and throws an error if they are not correct:

```matlab
function empl_profile1(empl_id, empl_info, healthplan, ... vacation)
```
validateattributes(empl_id, {'numeric'}, ...
   {'integer', 'nonempty'});
validateattributes(empl_info, {'struct'}, {'vector'});
validateattributes(healthplan, {'cell', 'char'}, ...
   {'vector'});
validateattributes(vacation, {'numeric'}, ...
   {'nonnegative', 'scalar'});

Call the empl_profile1 function, passing the expected argument types, and the example completes without error:

    empl_id = 51723;
    empl_info.name = 'John Miller';
    empl_info.address = '128 Forsythe St.';
    empl_info.town = 'Duluth'; empl_info.state='MN';

    empl_profile1(empl_id, empl_info, 'HCP Medical Plus', 14.3)

If you accidentally pass the argument values out of their correct sequence, MATLAB throws an error in response to the first argument that is not a match:

    empl_profile1(empl_id, empl_info, 14.3, 'HCP Medical Plus')

??? Error using ==> empl_profile1 at 6
Expected input to be one of these types:

    cell, char

Instead its type was double.

**Example 2**

Write a new function empl_profile2 that displays the function name, variable name, and position of the argument:

    function empl_profile2(empl_id, empl_info, healthplan, ...
       vacation)
validateattributes(empl_id, ...
   {'numeric'}, {'integer', 'nonempty'}, ...
   mfilename, 'Employee Identification', 1);

validateattributes(empl_info, ...
   {'struct'}, {'vector'}, ...
   mfilename, 'Employee Info', 2);

validateattributes(healthplan, ...
   {'cell', 'char'}, {'vector'}, ...
   mfilename, 'Health Plan', 3);

validateattributes(vacation, ...
   {'numeric'}, {'nonnegative', 'scalar'}, ...
   mfilename, 'Vacation Accrued', 4);

Call empl_profile2 with the argument values out of sequence. MATLAB throws an error that includes the name of the function validating the attributes, the name of the variable that was in error, and its position in the input argument list:

??? Error using ==> empl_profile2
Expected input number 3, Health Plan, to be one of these types:

   cell, char

Instead its type was double.

Error in ==> empl_profile2 at 12
validateattributes(healthplan, ...
Example 3

Write a new function `empl_profile2` that checks the input parameters with `inputParser`. Use `validateattributes` as the validating function for the `inputParser` methods:

```matlab
function empl_profile3(empl_id, varargin)
p = inputParser;

% Validate the input arguments.
addRequired(p, 'empl_id', ...
 @(x)validateattributes(x, {'numeric'}, {'integer'}));
addOptional(p, 'empl_info', '', ...
 @(x)validateattributes(x, {'struct'}, {'nonempty'}));
addParamValue(p, 'health', 'HCP Medical Plus', ...
 @(x)validateattributes(x, {'cell', 'char'}, ...
 {'vector'}));
addParamValue(p, 'vacation', [], ...
 @(x)validateattributes(x, {'numeric'}, ...
 {'nonnegative', 'scalar'}));
parse(p, empl_id, varargin{:});
p.Results

Call `empl_profile` using appropriate input arguments:

empl_info.name = 'John Miller';
empl_info.address = '128 Forsythe St.';
empl_info.town = 'Duluth'; empl_info.state='MN';

empl_profile(51723, empl_info, 'vacation', 14.3)

ans =
    empl_id: 51723
    empl_info: [1x1 struct]
        health: 'HCP Medical Plus'
        vacation: 14.3000
validateattributes

Call `empl_profile` using a character string where a structure is expected:

```matlab
empl_profile(51723, empl_info.name, 'vacation', 14.3)
```

??? Error using `==> empl_profile` at 12
Argument 'empl_info' failed validation with error:
Expected input to be one of these types:

- `struct`

Instead its type was `char`.

**Example 4**
Validate the size of a 4-by-2-by-6 array:

```matlab
x = rand(4,2,6);
validateattributes(x, {'numeric'}, {'size', [4,2,6]});
```

**Example 5**
The next two commands validate that the input value `x` lies within the specified range:

```matlab
x = uint8(50:10:200);
validateattributes(x, {'uint8'}, {'>=', 50, '<=', 200})
validateattributes(x, {'uint8'}, {'>=', 50, '<', 200})
```

??? Expected input to be an array with all of the values < 200.

**See Also**
`validatestring`, `is*`, `isa`, `inputParser`
Purpose

Check validity of text string

Syntax

\[
\text{validstr} = \text{validatestring} (\text{str}, \text{strarray})
\]
\[
\text{validstr} = \text{validatestring} (\text{str}, \text{strarray}, \text{position})
\]
\[
\text{validstr} = \text{validatestring} (\text{str}, \text{strarray}, \text{funname})
\]
\[
\text{validstr} = \text{validatestring} (\text{str}, \text{strarray}, \text{funname}, \text{varname})
\]
\[
\text{validstr} = \text{validatestring} (\text{str}, \text{strarray}, \text{funname}, \text{varname}, \text{position})
\]

Description

\[
\text{validstr} = \text{validatestring} (\text{str}, \text{strarray})
\]
checks the validity of text string \text{str}. If \text{str} matches one or more of the text strings in the cell array \text{strarray}, MATLAB returns the matching string in \text{validstr}. If \text{str} does not match any of the strings in \text{strarray}, MATLAB issues a formatted error message. MATLAB compares the strings without respect to letter case.

This table shows how \text{validatestring} determines what value to return. If multiple matches are found, \text{validatestring} returns the shortest matching string.

<table>
<thead>
<tr>
<th>Type of Match</th>
<th>Example — Match 'ball' with . . .</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact match</td>
<td>ball, barn, bell</td>
<td>ball</td>
</tr>
<tr>
<td>Partial match (leading characters)</td>
<td>balloon, barn</td>
<td>balloon</td>
</tr>
<tr>
<td>Multiple partial matches where each string is a subset of another</td>
<td>ball, ballo, balloo, balloon</td>
<td>ball</td>
</tr>
<tr>
<td>Multiple partial matches where strings are unique</td>
<td>balloon, ballet</td>
<td>Error</td>
</tr>
<tr>
<td>No match</td>
<td>barn, bell</td>
<td>Error</td>
</tr>
</tbody>
</table>

\[
\text{validstr} = \text{validatestring} (\text{str}, \text{strarray}, \text{position})
\]
checks the validity of text string \text{str} and, if the validation fails, displays an error message that includes the position of the failing variable in the function argument list. The \text{position} input must be a positive integer.
validstr = validatestring(str, strarray, funname) checks the validity of text string str and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname). The funname input must be a string enclosed in single quotation marks.

validstr = validatestring(str, strarray, funname, varname) checks the validity of text string str and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname) and the name of the variable being validated (varname). The funname and varname inputs must be strings enclosed in single quotation marks.

validstr = validatestring(str, strarray, funname, varname, position) checks the validity of text string str and, if the validation fails, displays an error message that includes the name of the function performing the validation (funname), the name of the variable being validated (varname), and the position of this variable in the function argument list (position). The funname and varname inputs must be strings enclosed in single quotation marks. The position input must be a positive integer.

Examples

Example 1

Use validatestring to find the word won in the cell array of strings:

```matlab
validatestring('won', {'wind', 'won', 'when'})
ans =
    won
```

Replace the word won with wonder in the string array. Because the leading characters of the input string and wonder are the same, validatestring finds a partial match between the two words and returns the full word wonder:

```matlab
validatestring('won', {'wind', 'wonder', 'when'})
ans =
    wonder
```
If there is more than one partial match, and each string in the array is a subset or superset of the others, `validatestring` returns the shortest matching string:

```plaintext
validatestring('wond', {'won', 'wonder', 'wonderful'})
ans =
    wonder
```

However, if each string in the array is not subset or superset of each other, MATLAB throws an error because there is no exact match and it is not clear which of the two partial matches should be returned:

```plaintext
validatestring('wond', {'won', 'wonder', 'wondrous'})
??? Error using ==> validatestring at 89
    Function VALIDATESTRING expected its input argument to match one of these strings:
    won, wonder, wondrous

    The input, 'wond', matched more than one valid string.
```

**Example 2**

In this example, the `get_flight_numbers` function returns the flight numbers for routes between two cities: a point of origin and point of destination. The function uses `validatestring` to see if the origin and destination are among those covered by the airline. If not, an error message is displayed:

```plaintext
function get_flight_numbers(origin, destination)
    % Only part of the airline's flight data is shown here.
    flights.chi2rio = [503, 196, 331, 373, 1475];
    flights.chi2par = [718, 9276, 172, 903, 7724 992, 1158];
    flights.chi2hon = [9193, 880, 471, 391];

    routes = {'Athens', 'Paris', 'Chicago', 'Sydney', ...
         'Cancun', 'London', 'Rio de Janeiro', 'Honolulu', ...
         'Rome', 'New York City'};
    orig = ''; dest = '';
% Does the airline cover these cities?
try
    orig = validatestring(origin, routes);
    dest = validatestring(destination, routes);
catch
    % If not covered, then display error message.
    if isempty(orig)
        fprintf(...
            'We have no flights with origin: %s.\n', ...
            origin)
    elseif isempty(dest)
        fprintf('%s%s%s.\n', 'We have no flights ', ...
            'with destination: ', destination)
    end
    return
end

% If covered, display the flights from 'orig' to 'dest'.
% reply = eval([... ...
    fprintf(...
        'Flights available from %s to %s are:\n', orig, dest)
    reply = eval([... ...
        ['flights.' lower(orig(1:3)) '2' lower(dest(1:3))]]);
    fprintf(' Flight %d\n', reply)

Enter a point of origin that is not covered by this airline:

    get_flight_numbers('San Diego', 'Rio de Janeiro')
ans =
    We have no flights with origin: San Diego.

Enter a destination that is misspelled:

    get_flight_numbers('Chicago', 'Reo de Janeiro')
ans =
    We have no flights with destination: Reo de Janeiro.

Enter a route that is covered:
get_flight_numbers('Chicago', 'Rio de Janeiro')
ans =
   Flights available from Chicago to Rio de Janeiro are:
   Flight 503
   Flight 196
   Flight 331
   Flight 373
   Flight 1475

Example 3

Rewrite the try-catch block of Example 2 by adding funname, varname, and position arguments to the call to validatestring and replacing the return statement with rethrow:

% See if the cities entered are covered by this airline.
try
    orig = validatestring(...
        origin, routes, mfilename, 'Flight Origin', 1);
    dest = validatestring(...
        destination, routes, mfilename, ...
        'Flight Destination', 2);
    catch e
        % If not covered, then display error message.
        if isempty(orig)
            fprintf(...
                'We have no flights with origin: %s.
                origin)
        elseif isempty(dest)
            fprintf('%s%s%s.
                with destination: ', dest)
        end
        rethrow(e);
    end

In response to the rethrow command, MATLAB displays an error message that includes the function name get_flight_numbers, the
failing variable name Flight Destination', and its position in the argument list, 2:

    get_flight_numbers('Chicago', 'Reo de Janeiro')
We have no flights with destination: Reo de Janeiro.

??? Error using ==> validatestring at 89
Function GET_FLIGHT_NUMBERS expected its input argument number 2, Flight Destination, to match one of these strings:

    Athens, Paris, Chicago, Sydney, Cancun, London, Rio de Janeiro, Honolulu, Rome

The input, 'Reo de Janeiro', did not match any of the valid strings.

Error in ==> get_flight_numbers at 17
    dest = validatestring(destination, routes, mfilename, 'destination', 2);

See Also

    validateattributes, is*, isa, inputParser
values (Map)

**Purpose**
Return values of containers.Map object

**Syntax**

- `v = values(M)`
- `v = values(M, keys)`

**Description**

- `v = values(M)` returns in cell array `v` the values that correspond to all keys in Map object `M`.

- `v = values(M, keys)` returns in cell array `v`, those values in Map object `M` that correspond to the keys specified by the `keys` argument.

Read more about Map Containers in the MATLAB Programming Fundamentals documentation.

**Examples**

Create a Map object of four US states and their capital cities:

```matlab
US_Capitals = containers.Map( ...  
    {'Georgia', 'Alaska', 'Vermont', 'Oregon'}, ...  
    {'Atlanta', 'Juneau', 'Montpelier', 'Salem'})
```

Find the capital cities of all states contained in the map:

```matlab
v = values(US_Capitals)
```

```matlab
v =  
     'Juneau'     'Atlanta'     'Salem'     'Montpelier'
```

Find the capital cities of selected states:

```matlab
v = values(US_Capitals, {'Oregon', 'Alaska'})
```

```matlab
v =  
     'Salem'     'Juneau'
```

**See Also**

containers.Map, keys(Map), size(Map), length(Map), isKey(Map), remove(Map), handle
Purpose
Vandermonde matrix

Syntax
A = vander(v)

Description
A = vander(v) returns the Vandermonde matrix whose columns are powers of the vector v, that is, A(i,j) = v(i)^(n-j), where n = length(v).

Examples
vander(1:.5:3)

ans =

1.0000 1.0000 1.0000 1.0000 1.0000
5.0625 3.3750 2.2500 1.5000 1.0000
16.0000 8.0000 4.0000 2.0000 1.0000
39.0625 15.6250 6.2500 2.5000 1.0000
81.0000 27.0000 9.0000 3.0000 1.0000

See Also
gallery
**Purpose**  
Variance

**Syntax**

\[ V = \text{var}(X) \]
\[ V = \text{var}(X,1) \]
\[ V = \text{var}(X,w) \]
\[ V = \text{var}(X,w,dim) \]

**Description**

\( V = \text{var}(X) \) returns the variance of \( X \) for vectors. For matrices, \( \text{var}(X) \) is a row vector containing the variance of each column of \( X \). For N-dimensional arrays, \( \text{var} \) operates along the first nonsingleton dimension of \( X \). The result \( V \) is an unbiased estimator of the variance of the population from which \( X \) is drawn, as long as \( X \) consists of independent, identically distributed samples.

\( \text{var} \) normalizes \( V \) by \( N-1 \) if \( N>1 \), where \( N \) is the sample size. This is an unbiased estimator of the variance of the population from which \( X \) is drawn, as long as \( X \) consists of independent, identically distributed samples. For \( N=1 \), \( V \) is normalized by \( N \).

\( V = \text{var}(X,1) \) normalizes by \( N \) and produces the second moment of the sample about its mean. \( \text{var}(X,0) \) is equivalent to \( \text{var}(X) \).

\( V = \text{var}(X,w) \) computes the variance using the weight vector \( w \). The length of \( w \) must equal the length of the dimension over which \( \text{var} \) operates, and its elements must be nonnegative. The elements of \( w \) must be positive. \( \text{var} \) normalizes \( w \) to sum of 1.

\( V = \text{var}(X,w,dim) \) takes the variance along the dimension \( \text{dim} \) of \( X \). Pass in 0 for \( w \) to use the default normalization by \( N-1 \), or 1 to use \( N \). The variance is the square of the standard deviation (STD).

**See Also**

corrcoef, cov, mean, median, std
**var (timeseries)**

**Purpose**
Variance of timeseries data

**Syntax**

```
var(ts)
var(ts,'PropertyName1',PropertyValue1,...)
```

**Description**

`ts_var = var(ts)` returns the variance of `ts.data`. When `ts.Data` is a vector, `ts_var` is the variance of `ts.Data` values. When `ts.Data` is a matrix, `ts_var` is a row vector containing the variance of each column of `ts.Data` (when `IsTimeFirst` is true and the first dimension of `ts` is aligned with time). For the N-dimensional `ts.Data` array, `var` always operates along the first nonsingleton dimension of `ts.Data`.

`ts_var = var(ts,'PropertyName1',PropertyValue1,...)` specifies the following optional input arguments:

- `'MissingData'` property has two possible values, 'remove' (default) or 'interpolate', indicating how to treat missing data during the calculation.
- `'Quality'` values are specified by an integer vector, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).
- `'Weighting'` property has two possible values, 'none' (default) or 'time'.
  When you specify 'time', larger time values correspond to larger weights.

**Examples**

The following example shows how to calculate the variance values of a multi-variate timeseries object.

1. Load a 24-by-3 data array.
   ```
   load count.dat
   ```

2. Create a timeseries object with 24 time values.
   ```
   count_ts = timeseries(count,[1:24],Name,'CountPerSecond')
   ```
3 Calculate the variance of each data column for this \texttt{timeseries}
object.

\begin{verbatim}
    var(count_ts)
    ans =

    1.0e+003 *
    0.6437   1.7144   4.6278
\end{verbatim}

The variance is calculated independently for each data column in the
\texttt{timeseries} object.

\textbf{See Also} \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ iqr (timeseries), mean (timeseries), median (timeseries), std (timeseries), timeseries
**Purpose**
Variable length input argument list

**Syntax**

```matlab
function y = bar(varargin)
```

**Description**

The `varargin` statement is used only inside a function M-file to contain optional input arguments passed to the function. The `varargin` argument must be declared as the last input argument to a function, collecting all the inputs from that point onwards. In the declaration, `varargin` must be lowercase.

**Examples**

**Example 1**

Write an M-file function that displays the expected and optional arguments you pass to it

```matlab
function vartest(argA, argB, varargin)

optargin = size(varargin,2);
stdargin = nargin - optargin;

fprintf('Number of inputs = %d
', nargin)

fprintf(' Inputs from individual arguments(%d):\n', ... stdargin)
if stdargin >= 1
    fprintf('   %d
', argA)
end
if stdargin == 2
    fprintf('   %d
', argB)
end

fprintf(' Inputs packaged in varargin(%d):
', optargin)
for k= 1 : size(varargin,2)
    fprintf('   %d\n', varargin{k})
end
```
Call this function and observe that the MATLAB software extracts those arguments that are not individually-specified from the varargin cell array:

\[ \text{vartest}(10,20,30,40,50,60,70) \]
\[ \text{Number of inputs} = 7 \]
\[ \text{Inputs from individual arguments(2):} \]
\[ 10 \]
\[ 20 \]
\[ \text{Inputs packaged in varargin(5):} \]
\[ 30 \]
\[ 40 \]
\[ 50 \]
\[ 60 \]
\[ 70 \]

**Example 2**

The function

\[ \text{function myplot(x,varargin)} \]
\[ \text{plot(x,varargin{:})} \]

collects all the inputs starting with the second input into the variable varargin. myplot uses the comma-separated list syntax varargin{:} to pass the optional parameters to plot. The call

\[ \text{myplot(sin(0:.1:1),'color',[.5 .7 .3],'linestyle',':')} \]

results in varargin being a 1-by-4 cell array containing the values 'color', [.5 .7 .3], 'linestyle', and ':'.

**See Also**

varargout, nargin, nargout, nargchk, nargoutchk, inputname
Purpose
Variable length output argument list

Syntax
function varargout = foo(n)

Description
function varargout = foo(n) returns a variable number of arguments from function foo.m.

The varargout statement is used only inside a function M-file to contain the optional output arguments returned by the function. The varargout argument must be declared as the last output argument to a function, collecting all the outputs from that point onwards. In the declaration, varargout must be lowercase.

Examples
The function

```matlab
function [s,varargout] = mysize(x)
nout = max(nargout,1)-1;
s = size(x);  
for k=1:nout, varargout(k) = {s(k)}; end
```

returns the size vector and, optionally, individual sizes. So

```matlab
[s,rows,cols] = mysize(rand(4,5));
```

returns \( s = [4 \ 5], \) rows = 4, cols = 5.

See Also
varargin, nargin, nargout, nargchk, nargoutchk, inputname
Purpose  Vectorize expression

Syntax

vectorize(s)
vectorize(fun)

Description

vectorize(s) where s is a string expression, inserts a . before any ^, *, or / in s. The result is a character string.

vectorize(fun) when fun is an inline function object, vectorizes the formula for fun. The result is the vectorized version of the inline function.

See Also  inline, cd, dbtype, delete, dir, partialpath, path, what, who
Purpose
Version information for MathWorks products

GUI Alternatives
As an alternative to the ver function, select Help > About in any tool that has a Help menu.

Syntax
```matlab
ver
ver product
v = ver('product')
```

Description
ver displays a header containing the current MathWorks product family version number, license number, operating system, and version of Sun Microsystems JVM software for the MATLAB product. This is followed by the version numbers for MATLAB, Simulink, if installed, and all other installed MathWorks products.

ver product displays the MathWorks product family header information followed by the current version number for product. The name product corresponds to the directory name that holds the Contents.m file for that product. For example, Contents.m for the Control System Toolbox product resides in the control directory. You therefore use ver control to obtain the version of this toolbox.

v = ver('product') returns the version information to structure array, v, having fields Name, Version, Release, and Date.

Remarks
To use ver with your own product, the first two lines of the Contents.m file for the product must be of the form

```matlab
% Toolbox Description
% Version xxx dd-mmm-yyyy
```

Do not include any spaces in the date and use a two-character day; that is, use 02-Sep-2008 instead of 2-Sep-2008.

Examples
Using R2009a, return version information for MathWorks products, and specifically the Control System Toolbox product by typing

```matlab
ver control
```
MATLAB returns

Return version information for the Control System Toolbox product in a structure array, v.

```matlab
v = ver('control')
v =
```

```matlab
Name: 'Control System Toolbox'
Version: '8.3'
Release: '(R2009a)'
Date: '24-Jan-2009'
```

Display version information for MathWorks 'Real-Time' products:

```matlab
v = ver;
for k=1:length(v)
    if strfind(v(k).Name, 'Real-Time')
        disp(sprintf('%s, Version %s', ...
            v(k).Name, v(k).Version))
    end
end
```

```matlab
Real-Time Windows Target, Version 3.3
Real-Time Workshop, Version 7.3
Real-Time Workshop Embedded Coder, Version 5.3
```

**See Also**

computer, help, hostid, license, verlessthan, version, whatsnew

**Help > Check for Updates** in the MATLAB desktop.
Source control actions (Windows platforms)

As an alternative to the `verctrl` function, use **Source Control** in the **File** menu of the Editor, the Simulink product, or the Stateflow product, or in the context menu of the Current Directory browser.

Syntax

```matlab
verctrl('action',{"filename1","filename2",...},0)
result=verctrl('action',{"filename1","filename2",...},0)
verctrl('action','filename',0)
result=verctrl('isdiff','filename',0)
list = verctrl('all_systems')
```

Description

`verctrl('action',{"filename1","filename2",...},0)` performs the source control operation specified by `action` for a single file or multiple files. Enter one file as a string; specify multiple files using a cell array of strings. Use the full paths for each filename and include the extensions. Specify 0 as the last argument. Complete the resulting dialog box to execute the operation; for details about the dialog boxes, see the topic “Source Control Interface on Microsoft Windows” in the MATLAB Desktop Tools and Development Environment documentation. Available values for `action` are as follows:

<table>
<thead>
<tr>
<th>Action Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'add'</td>
<td>Adds files to the source control system. Files can be open in the Editor or closed when added.</td>
</tr>
<tr>
<td>'checkin'</td>
<td>Checks files into the source control system, storing the changes and creating a new version.</td>
</tr>
<tr>
<td>'checkout'</td>
<td>Retrieves files for editing.</td>
</tr>
<tr>
<td>'get'</td>
<td>Retrieves files for viewing and compiling, but not editing. When you open the files, they are labeled as read-only.</td>
</tr>
<tr>
<td>'history'</td>
<td>Displays the history of files.</td>
</tr>
<tr>
<td>action Argument</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td>'remove'</td>
<td>Removes files from the source control system. It does not delete the files from disk, but only from the source control system.</td>
</tr>
<tr>
<td>'runsc'</td>
<td>Starts the source control system. The filename can be an empty string.</td>
</tr>
<tr>
<td>'uncheckout'</td>
<td>Cancels a previous checkout operation and restores the contents of the selected files to the precheckout version. All changes made to the files since the checkout are lost.</td>
</tr>
</tbody>
</table>

result=verctrl('action',{'filename1','filename2',....},0)
performs the source control operation specified by 'action' on a single file or multiple files. The action can be any one of: 'add', 'checkin', 'checkout', 'get', 'history', or 'undocheckout'. result is a logical 1 (true) when you complete the operation by clicking OK in the resulting dialog box, and is a logical 0 (false) when you abort the operation by clicking Cancel in the resulting dialog box.

verctrl('action','filename',0) performs the source control operation specified by 'action' for a single file. Use the full pathname for 'filename'. Specify 0 as the last argument. Complete any resulting dialog boxes to execute the operation. Available values for 'action' are as follows:

<table>
<thead>
<tr>
<th>action Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'showdiff'</td>
<td>Displays the differences between a file and the latest checked in version of the file in the source control system.</td>
</tr>
<tr>
<td>'properties'</td>
<td>Displays the properties of a file.</td>
</tr>
</tbody>
</table>
result=verctrl('isdiff','filename',0) compares filename with the latest checked in version of the file in the source control system. result is a logical 1 (true) when the files are different, and is a logical 0 (false) when the files are identical. Use the full path for 'filename'. Specify 0 as the last argument.

list = verctrl('all_systems') displays in the Command Window a list of all source control systems installed on your computer.

Examples

**Check In a File**

Check in D:\file1.ext to the source control system.

    result = verctrl('checkin','D:\file1.ext', 0)

This opens the Check in file(s) dialog box. Click **OK** to complete the check in. MATLAB displays result = 1, indicating the checkin was successful.

**Add Files to the Source Control System**

Add D:\file1.ext and D:\file2.ext to the source control system.

    verctrl('add',{'D:\file1.ext','D:\file2.ext'}, 0)

This opens the Add to source control dialog box. Click **OK** to complete the operation.

**Display the Properties of a File**

Display the properties of D:\file1.ext.

    verctrl('properties','D:\file1.ext', 0)

This opens the source control properties dialog box for your source control system. The function is complete when you close the properties dialog box.
Show Differences for a File

To show the differences between the version of file1.ext that you just edited and saved, with the last version in source control, run

```
verctrl('showdiff','D:\file1.ext',0)
```

MATLAB displays differences dialog boxes and results specific to your source control system. After checking in the file, if you run this statement again, MATLAB displays

```matlab
??? The file is identical to latest version under source control.
```

List All Installed Source Control Systems

To view all of the source control systems installed on your computer, type

```
list = verctrl ('all_systems')
```

MATLAB displays all the source control systems currently installed on your computer. For example:

```
list =
'Microsoft Visual SourceSafe'
'ComponentSoftware RCS'
```

See Also

checkin, checkout, undocheckout, cmopts

“Source Control Interface on Microsoft Windows” in MATLAB Desktop Tools and Development Environment documentation
Purpose

Compare toolbox version to specified version string

Syntax

verLessThan(toolbox, version)

Description

verLessThan(toolbox, version) returns logical 1 (true) if the version of the toolbox specified by the string toolbox is older than the version specified by the string version, and logical 0 (false) otherwise. Use this function when you want to write code that can run across multiple versions of the MATLAB software, when there are differences in the behavior of the code in the different versions.

The toolbox argument is a string enclosed within single quotation marks that contains the name of a MATLAB toolbox directory. The version argument is a string enclosed within single quotation marks that contains the version to compare against. This argument must be in the form major[.minor[.revision]], such as 7, 7.1, or 7.0.1. If toolbox does not exist, MATLAB generates an error.

To specify toolbox, find the directory that holds the Contents.m file for the toolbox and use that directory name. To see a list of all toolbox directory names, enter the following statement in the MATLAB Command Window:

```matlab
dir([matlabroot '/toolbox'])
```

Remarks

The verLessThan function is available with MATLAB Version 7.4 and subsequent versions. If you are running a version of MATLAB prior to 7.4, you can download the verLessThan M-file from the following MathWorks Technical Support solution. You must be running MATLAB Version 6.0 or higher to use this M-file:

http://www.mathworks.com/support/solutions/data/1-38LI61.html?solution=1

Examples

These examples illustrate usage of the verLessThan function.

**Example 1 – Checking For the Minimum Required Version**

```matlab
if verLessThan('simulink', '4.0')
    error('Simulink 4.0 or higher is required.');
```
Example 2 – Choosing Which Code to Run

if verLessThan('matlab', '7.0.1')
  % -- Put code to run under MATLAB 7.0.0 and earlier here --
else
  % -- Put code to run under MATLAB 7.0.1 and later here --
end

Example 3 – Looking Up the Directory Name

Find the name of the Data Acquisition Toolbox directory:

dir([matlabroot '/toolbox/d*'])

daq database des distcomp dotnetbuilder
dastudio datafeed dials dml dspblks

Use the toolbox directory name, daq, to compare the Data Acquisition Toolbox software version that MATLAB is currently running against version number 3:

verLessThan('daq', '3')
ans =
  1

See Also

ver, version, license, ispc, isunix, ismac, dir
**version**

**Purpose**  
Version number for MATLAB

**GUI Alternatives**  
As an alternative to the `version` function, select **Help > About MATLAB** in the MATLAB desktop.

**Syntax**  
```
version
v = version
version option
v = version('option')
```

**Description**  
`version` displays the MathWorks product family version.  

`v = version` returns the MathWorks product family version number in string `v`.  

`version option` displays the following additional information about the version. You can specify no more than one option in a `version` command.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-date</code></td>
<td>Release date</td>
</tr>
<tr>
<td><code>-java</code></td>
<td>Sun Microsystems JVM software version used by MATLAB</td>
</tr>
<tr>
<td><code>-release</code></td>
<td>Release number</td>
</tr>
</tbody>
</table>

`v = version('option')` returns in string `v` the information displayed in response to the syntax shown above. You can only specify no more than one option when using this syntax.

**Remarks**  
On Microsoft Windows and UNIX\(^{30}\) platforms, MATLAB includes JVM software and uses that version. If you use the MATLAB interface to

---

30. UNIX is a registered trademark of The Open Group in the United States and other countries.
Sun Microsystems Java software and the Java classes you want to use require a different version of the JVM software than that provided with MATLAB, it is possible to run MATLAB with a different version of JVM software. For details, see “Using a Different Version of JVM Software”.

On the Apple Macintosh platform, MATLAB does not include JVM software, but uses JVM software provided with Mac OS X.

**Examples**

```matlab
d = version('-date')
d =
    January 24, 2009
```

Run the following command in MATLAB Version 7.8 (R2009a):

```matlab
['Release R' version('-release')]
```

```matlab
ans =
    Release R2009a
```

**See Also**

computer, ver, verlessthan, whatsnew

**Help > Check for Updates** in the MATLAB desktop.
**Purpose**
Concatenate arrays vertically

**Syntax**

\[
C = \text{vertcat}(A_1, A_2, \ldots)
\]

**Description**
\[
C = \text{vertcat}(A_1, A_2, \ldots)
\] vertically concatenates matrices \(A_1, A_2, \) and so on. All matrices in the argument list must have the same number of columns.

\text{vertcat} concatenates \(N\)-dimensional arrays along the first dimension. The remaining dimensions must match.

MATLAB calls \(C = \text{vertcat}(A_1, A_2, \ldots)\) for the syntax \(C = [A_1; A_2; \ldots]\) when any of \(A_1, A_2, \) etc. is an object.

**Examples**
Create a 5-by-3 matrix, \(A\), and a 3-by-3 matrix, \(B\). Then vertically concatenate \(A\) and \(B\).

\[
A = \text{magic}(5); \quad \% \text{Create 5-by-3 matrix, } A
A(:, 4:5) = []
\]

\[
A =
\begin{bmatrix}
17 & 24 & 1 \\
23 & 5 & 7 \\
4 & 6 & 13 \\
10 & 12 & 19 \\
11 & 18 & 25 \\
\end{bmatrix}
\]

\[
B = \text{magic}(3)*100 \quad \% \text{Create 3-by-3 matrix, } B
\]

\[
B =
\begin{bmatrix}
800 & 100 & 600 \\
300 & 500 & 700 \\
400 & 900 & 200 \\
\end{bmatrix}
\]
C = vertcat(A,B) % Vertically concatenate A and B

C =

17    24    1
23     5     7
 4     6    13
10    12    19
11    18    25
800   100   600
300   500   700
400   900   200

See Also    horzcat, cat
Purpose
Vertical concatenation of timeseries objects

Syntax
\( ts = \text{vertcat}(ts1, ts2, ...) \)

Description
\( ts = \text{vertcat}(ts1, ts2, ...) \) performs
\[ ts = [ts1; ts2; ...] \]

This operation appends timeseries objects. The time vectors must not overlap. The last time in ts1 must be earlier than the first time in ts2. The data sample size of the timeseries objects must agree.

See Also
timeseries
Purpose  Vertical concatenation for tscollection objects

Syntax  \( tsc = \text{vertcat}(tsc1,tsc2,...) \)

Description  \( tsc = \text{vertcat}(tsc1,tsc2,...) \) performs
\( tsc = [tsc1;tsc2;...] \)

This operation appends tscollection objects. The time vectors must not overlap. The last time in tsc1 must be earlier than the first time in tsc2. All tscollection objects to be concatenated must have the same timeseries members.

See Also  horzcat (tscollection), tscollection
**TriRep.vertexAttachments**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Return simplices attached to specified vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>$SI = vertexAttachments(TR, VI)$</td>
</tr>
<tr>
<td>Description</td>
<td>$SI = vertexAttachments(TR, VI)$ returns the vertex-to-simplex information for the specified vertices $VI$. In relation to 2-D triangulations, if the triangulation has a consistent orientation the triangles in each cell will be ordered consistently around each vertex.</td>
</tr>
<tr>
<td>Inputs</td>
<td>$TR$ Triangulation representation $VI$ $VI$ is a column vector of indices into the array of points representing the vertex coordinates, $TR.X$. The simplices associated with vertex $i$ are the $i$'th entry in the cell array. If $VI$ is not specified the vertex-simplex information for the entire triangulation is returned.</td>
</tr>
<tr>
<td>Outputs</td>
<td>$SI$ Cell array of indices of the simplices attached to a vertex. A cell array is used to store the information because the number of simplices associated with each vertex can vary. The simplices associated with vertex $i$ are in the $i$'th entry in the cell array $SI$.</td>
</tr>
<tr>
<td>Definitions</td>
<td>A simplex is a triangle/tetrahedron or higher dimensional equivalent.</td>
</tr>
<tr>
<td>Examples</td>
<td><strong>Example 1</strong></td>
</tr>
<tr>
<td></td>
<td>Load a 2-D triangulation and use TriRep to compute the vertex-to-triangle relations.</td>
</tr>
<tr>
<td></td>
<td><code>load trimesh2d</code></td>
</tr>
<tr>
<td></td>
<td>Find the indices of the tetrahedra attached to the first vertex:</td>
</tr>
</tbody>
</table>
Tv = vertexAttachments(trep, 1)
Tv{:}

**Example 2**

Perform a direct query of a 2-D triangulation created using DelaunayTri.

```matlab
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);

Find the triangles attached to vertex 5:

```matlab
t = vertexAttachments(dt, 5);
```

Plot the triangulation:

```matlab
triplot(dt);
hold on;
```

Plot the triangles attached to vertex 5 (in red):

```matlab
triplot(dt(t{:},:),x,y,'Color','r');
hold off;
```
**See Also**  
DelaunayTri
**Purpose**

Viewpoint specification

**Syntax**

```plaintext
view(az,el)
view([x,y,z])
view(2)
view(3)
view(ax,...)
view(T)
[az,el] = view
T = view
```

**Description**

The position of the viewer (the viewpoint) determines the orientation of the axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,el)` and `view([az,el])` set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the z-axis as measured in degrees from the negative y-axis. Positive values indicate counterclockwise rotation of the viewpoint. `el` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the viewpoint to the Cartesian coordinates `x`, `y`, and `z`. The magnitude of `(x,y,z)` is ignored.

`view(2)` sets the default two-dimensional view, `az = 0`, `el = 90`.

`view(3)` sets the default three-dimensional view, `az = 37.5`, `el = 30`.

`view(ax,...)` uses axes `ax` instead of the current axes.

`view(T)` sets the view according to the transformation matrix `T`, which is a 4-by-4 matrix such as a perspective transformation generated by `viewmtx`.

`[az,el] = view` returns the current azimuth and elevation.

`T = view` returns the current 4-by-4 transformation matrix.
Remarks

Azimuth is a polar angle in the $x$-$y$ plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the $x$-$y$ plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.

Examples

View the object from directly overhead.

```matlab
az = 0;
el = 90;
view(az, el);
```

Set the view along the $y$-axis, with the $x$-axis extending horizontally and the $z$-axis extending vertically in the figure.

```matlab
view([0 0]);
```

Rotate the view about the $z$-axis by $180^\circ$. 

2-4112
az = 180;
el = 90;
view(az, el);

See Also

viewmtx, hgtransform, rotate3d

“Camera Viewpoint” on page 1-106 for related functions

Axes graphics object properties CameraPosition, CameraTarget, CameraViewAngle, Projection

Defining the View for more information on viewing concepts and techniques

Transforming Objects for information on moving and scaling objects in groups
Purpose

View transformation matrices

Syntax

viewmtx
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,phi,xc)

Description

viewmtx computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

T = viewmtx(az,el) returns an orthographic transformation matrix corresponding to azimuth az and elevation el. az is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. el is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

view(az,el)
T = view

but does not change the current view.

T = viewmtx(az,el,phi) returns a perspective transformation matrix. phi is the perspective viewing angle in degrees. phi is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

<table>
<thead>
<tr>
<th>Phi</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 degrees</td>
<td>Orthographic projection</td>
</tr>
<tr>
<td>10 degrees</td>
<td>Similar to telephoto lens</td>
</tr>
<tr>
<td>25 degrees</td>
<td>Similar to normal lens</td>
</tr>
<tr>
<td>60 degrees</td>
<td>Similar to wide-angle lens</td>
</tr>
</tbody>
</table>

You can use the matrix returned to set the view transformation with view(T). The 4-by-4 perspective transformation matrix transforms four-dimensional homogeneous vectors into unnormalized vectors of the
form \((x,y,z,w)\), where \(w\) is not equal to 1. The \(x\)- and \(y\)-components of the normalized vector \((x/w, y/w, z/w, 1)\) are the desired two-dimensional components (see example below).

\[
T = \text{viewmtx}(az, el, phi, xc)\]
returns the perspective transformation matrix using \(xc\) as the target point within the normalized plot cube (i.e., the camera is looking at the point \(xc\)). \(xc\) is the target point that is the center of the view. You specify the point as a three-element vector, \(xc = [xc, yc, zc]\), in the interval \([0,1]\). The default value is \(xc = [0, 0, 0]\).

**Remarks**

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, \([x, y, z, 1]\) is the four-dimensional vector corresponding to the three-dimensional point \([x, y, z]\).

**Examples**

Determine the projected two-dimensional vector corresponding to the three-dimensional point \((0.5, 0.0, -3.0)\) using the default view direction. Note that the point is a column vector.

\[
A = \text{viewmtx}(-37.5, 30); \\
x4d = [.5 0 -3 1]'; \\
x2d = A*x4d; \\
x2d = x2d(1:2) \\
x2d = \\
0.3967 \\
-2.4459
\]

Vectors that trace the edges of a unit cube are

\[
x = [0 1 1 0 0 0 1 1 0 0 1 1 1 1 0 0]; \\
y = [0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1]; \\
z = [0 0 0 0 0 1 1 1 1 1 0 0 1 1 0 0];
\]

Transform the points in these vectors to the screen, then plot the object.

\[
A = \text{viewmtx}(-37.5, 30); \\
[m,n] = \text{size}(x); \\
x4d = [x(:,), y(:,), z(:,), ones(m*n,1)]';
\]
Use a perspective transformation with a 25 degree viewing angle:

```matlab
A = viewmtx(-37.5,30,25);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4)   % Normalize
x2d =
```

```matlab
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
plot(x2,y2)
```
Transform the cube vectors to the screen and plot the object:

```matlab
A = viewmtx(-37.5,30,25);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)./x2d(4,:);
y2(:) = x2d(2,:)./x2d(4,:);
plot(x2,y2)
```


See Also

view, hgtransform

“Camera Viewpoint” on page 1-106 for related functions

Defining the View for more information on viewing concepts and techniques
Purpose

Compare two text files, MAT-Files, or binary files

GUI Alternative

As an alternative to the visdiff function, select Desktop > File and Directory Comparisons, and then in the File and Directory Comparisons tool select File > New File Comparison.

Syntax

visdiff('fname1', 'fname2')
visdiff('fname1', 'fname2', showchars)
S = visdiff('fname1', 'fname2')

Description

visdiff('fname1', 'fname2') opens the File and Directory Comparisons tool and presents the differences between the two files. The two files must be on the MATLAB path, or you must provide the full path for each file.

visdiff('fname1', 'fname2', showchars) opens the File and Directory Comparisons tool with the width of each column in the display set to showchars characters wide. MATLAB ignores the showchars option when you compare two binary files or two MAT-Files.

S = visdiff('fname1', 'fname2') creates an HTML report describing the differences between the two files and returns it in the string S.

Remarks

MATLAB supports displaying the differences in the File and Directory Comparisons tool only if Java software is installed. However, MATLAB supports assigning the HTML report to a string, even if the Java software is not installed.

Examples

Compare Two Text Files in the Current Directory

For this example, copy the sample files, lengthofline.m and lengthofline2.m to your current directory as described in “Comparing Files and Directories”, and then type:

visdiff('lengthofline.m', 'lengthofline2.m', 30)
The File and Directory Comparisons tool opens and presents the differences between the two text files with the width of the left column set to 30.
Go to first difference

1 function [len, dims] = lengthofline x function [len, dims] = lengthofline

2 %LENGTHOFLINE Calculates the length of a line.

3 % LENGTHOFLINE Calculates the length of a

4 % input, and returns its len.

5 % input, and returns its len.

6 % dependent on the number of.

7 % dependent on the number of

8 % [LEN, DIM] = LENGTHOFLINE(H).

9 % [LEN, DIM] = LENGTHOFLINE(H)

10 % 2D or 3D by returning either

11 % 2D or 3D by returning either

12 % plane parallel to a coordin.

13 % plane parallel to a coordin.

14 % If HLINE is a matrix of length.

15 % If HLINE is a matrix of length

16 % Example:

17 % Example:

18 % figure; h2 = plot3(1:1); figure; h2 = plot3(1:1)

19 % hold on; h1 = plot(1:1); hold on; h1 = plot(1:1)

20 % [len, dim] = lengthofline.

21 % [len, dim] = lengthofline

22 % Find input indices that are.

23 % Find input indices that are

24 nohandle = ~ishandle(hline); nohandle = ~ishandle(hline);

25 > notline = false(size(hline));
**Compare Two MAT-Files Off the MATLAB Path**

If you enter the following command, the File and Directory Comparisons tool opens and presents the differences between the two MAT-Files.

```
visdiff('C:\Program Files\MATLAB\R2009a\toolbox\matlab\demos\gatlin.mat', ...
'C:\Program Files\MATLAB\R2009a\toolbox\matlab\demos\gatlin2.mat')
```

**Compare Two Binary Files on the MATLAB Path**

Add the directory containing two MEX-files to the MATLAB path, and then compare the files, by issuing the following commands:

```
addpath([matlabroot '\extern\examples\shrlib'])
```
visdiff('shrlibsample.mexw32', 'yprime.mexw32')

The File and Directory Comparisons tool opens and indicates that the files are different, but does not provide details about the differences.

The files are **different**. MATLAB cannot display the differences between files of these types.

**See Also**  
“Comparing Files and Directories”
**Purpose**

Coordinate and color limits for volume data

**Syntax**

\[
\text{lims} = \text{volumebounds}(X,Y,Z,V) \\
\text{lims} = \text{volumebounds}(X,Y,Z,U,V,W) \\
\text{lims} = \text{volumebounds}(V), \text{lims} = \text{volumebounds}(U,V,W)
\]

**Description**

\[
\text{lims} = \text{volumebounds}(X,Y,Z,V) \text{ returns the } x, y, z, \text{ and color limits of the current axes for scalar data. lims is returned as a vector:}
\]

\[
[x_{min} \ x_{max} \ y_{min} \ y_{max} \ z_{min} \ z_{max} \ c_{min} \ c_{max}]
\]

You can pass this vector to the axis command.

\[
\text{lims} = \text{volumebounds}(X,Y,Z,U,V,W) \text{ returns the } x, y, \text{ and z limits of the current axes for vector data. lims is returned as a vector:}
\]

\[
[x_{min} \ x_{max} \ y_{min} \ y_{max} \ z_{min} \ z_{max}]
\]

\[
\text{lims} = \text{volumebounds}(V), \text{lims} = \text{volumebounds}(U,V,W) \text{ assumes } X, Y, \text{ and } Z \text{ are determined by the expression}
\]

\[
[X \ Y \ Z] = \text{meshgrid}(1:n,1:m,1:p)
\]

where \([m \ n \ p] = \text{size}(V)\).

**Examples**

This example uses \text{volumebounds} to set the axis and color limits for an isosurface generated by the \text{flow} function.

\[
[x \ y \ z \ v] = \text{flow}; \\
p = \text{patch(isosurface}(x,y,z,v,-3)); \\
\text{isonormals}(x,y,z,v,p) \\
\text{daspect([1 1 1])} \\
\text{isocolors}(x,y,z,\text{flipdim}(v,2),p) \\
\text{shading interp} \\
\text{axis(}\text{volumebounds}(x,y,z,v)) \\
\text{view}(3) \\
\text{camlight} \\
\text{lighting phong}
\]
See Also
isosurface, streamslice

“Volume Visualization” on page 1-108 for related functions
**Purpose**
Voronoi diagram

**Syntax**
voronoi(x,y)
voronoi(x,y,TRI)
voronoi(X,Y,options)
voronoi(AX,...)
voronoi(...,'LineSpec')
h = voronoi(...)
[vx,vy] = voronoi(...)

**Definition**
Consider a set of coplanar points \( P \). For each point \( P_x \) in the set \( P \), you can draw a boundary enclosing all the intermediate points lying closer to \( P_x \) than to other points in the set \( P \). Such a boundary is called a Voronoi polygon, and the set of all Voronoi polygons for a given point set is called a Voronoi diagram.

**Description**
voronoi(x,y) plots the bounded cells of the Voronoi diagram for the points x,y. Lines-to-infinity are approximated with an arbitrarily distant endpoint.

voronoi(x,y,TRI) uses the triangulation TRI instead of computing it via delaunay.

voronoi(X,Y,options) specifies a cell array of strings to be used as options in Qhull via delaunay.

If options is [], the default delaunay options are used. If options is {''}, no options are used, not even the default.

voronoi(AX,...) plots into AX instead of gca.

voronoi(...,'LineSpec') plots the diagram with color and line style specified.

h = voronoi(...) returns, in h, handles to the line objects created.

[vx,vy] = voronoi(...) returns the finite vertices of the Voronoi edges in vx and vy so that plot(vx,vy,'-',x,y,'.') creates the Voronoi diagram. The lines-to-infinity are the last columns of vx and
vy. To ensure the lines-to-infinity do not affect the settings of the axis limits, use the commands:

```matlab
h = plot(VX,VY,'-',[X,Y,'r.']);
set(h(1:end-1),'xliminclude','off','yliminclude','off')
```

**Note** For the topology of the Voronoi diagram, i.e., the vertices for each Voronoi cell, use `voronoin`.

```matlab
[v,c] = voronoin([x(:) y(:)])
```

### Visualization

Use one of these methods to plot a Voronoi diagram:

- If you provide no output argument, `voronoi` plots the diagram. See Example 1.
- To gain more control over color, line style, and other figure properties, use the syntax `[vx,vy] = voronoi(...)`. This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the `plot` function. See Example 2.
- To fill the cells with color, use `voronoin` with `n=2` to get the indices of each cell, and then use `patch` and other plot functions to generate the figure. Note that `patch` does not fill unbounded cells with color. See Example 3.

### Examples

**Example 1**

This code uses the `voronoi` function to plot the Voronoi diagram for 10 randomly generated points.

```matlab
rand('state',5);
x = rand(1,10); y = rand(1,10);
voronoi(x,y)
```
Example 2

This code uses the vertices of the finite Voronoi edges to plot the Voronoi diagram for the same 10 points.

```
rand('state',5);
x = rand(1,10); y = rand(1,10);
[vx, vy] = voronoi(x,y);
plot(x,y,'r+',vx,vy,'b-'); axis equal
```

Note that you can add this code to get the figure shown in Example 1.

```
xlim([min(x) max(x)])
```
Example 3

This code uses `voronoin` and `patch` to fill the bounded cells of the same Voronoi diagram with color.

```matlab
rand('state',5);
x=rand(10,2);
[v,c]=voronoin(x);
for i = 1:length(c)
    if all(c{i}~=1) % If at least one of the indices is 1,
        % then it is an open region and we can't
        % patch that.
```

```
Algorithm
If you supply no triangulation TRI, the voronoi function performs a Delaunay triangulation of the data that uses Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also
convhull, delaunay, LineSpec, plot, voronoin

Reference
Purpose

Voronoi diagram

Syntax

\[[V, R] = \text{voronoiDiagram}(DT)\]

Description

\[[V, R] = \text{voronoiDiagram}(DT)\] returns the vertices \(V\) and regions \(R\) of the Voronoi diagram of the points \(DT.X\). The region \(R\{i\}\) is a cell array of indices into \(V\) that represents the Voronoi vertices bounding the region. The Voronoi region associated with the \(i^{th}\) point, \(DT.X(i)\) is \(R\{i\}\). For 2-D, vertices in \(R\{i\}\) are listed in adjacent order, i.e. connecting them will generate a closed polygon (Voronoi diagram). For 3-D the vertices in \(R\{i\}\) are listed in ascending order.

The Voronoi regions associated with points that lie on the convex hull of \(DT.X\) are unbounded. Bounding edges of these regions radiate to infinity. The vertex at infinity is represented by the first vertex in \(V\).

Inputs

\(DT\)

Delaunay triangulation.

Outputs

\(V\)

\(\text{numv-by-ndim}\) matrix representing the coordinates of the Voronoi vertices, where \(\text{numv}\) is the number of vertices and \(\text{ndim}\) is the dimension of the space where the points reside.

\(R\)

Vector cell array of \(\text{length}(DR.X)\), representing the Voronoi cell associated with each point.

Definitions

The Voronoi diagram of a discrete set of points \(X\) decomposes the space around each point \(X(i)\) into a region of influence \(R\{i\}\). Locations within the region are closer to point \(i\) than any other point. The region of influence is called the Voronoi region. The collection of all the Voronoi regions is the Voronoi diagram.
The convex hull of a set of points $X$ is the smallest convex polygon (or polyhedron in higher dimensions) containing all of the points of $X$.

**Examples**

Compute the Voronoi Diagram of a set of points:

```matlab
X = [ 0.5  0
      0  0.5
     -0.5 -0.5
     -0.2 -0.1
     -0.1  0.1
      0.1 -0.1
      0.1  0.1 ]
dt = DelaunayTri(X)
[V,R] = voronoiDiagram(dt)
```

**See Also**

voronoi
voronoin
**Purpose**

N-D Voronoi diagram

**Syntax**

\[
[V,C] = \text{voronoin}(X)
\]

\[
[V,C] = \text{voronoin}(X, \text{options})
\]

**Description**

\[
[V,C] = \text{voronoin}(X)
\] returns Voronoi vertices \(V\) and the Voronoi cells \(C\) of the Voronoi diagram of \(X\). \(V\) is a \(numv\)-by-\(n\) array of the \(numv\) Voronoi vertices in \(n\)-dimensional space, each row corresponds to a Voronoi vertex. \(C\) is a vector cell array where each element contains the indices into \(V\) of the vertices of the corresponding Voronoi cell. \(X\) is an \(m\)-by-\(n\) array, representing \(m\) \(n\)-dimensional points, where \(n > 1\) and \(m \geq n+1\).

The first row of \(V\) is a point at infinity. If any index in a cell of the cell array is 1, then the corresponding Voronoi cell contains the first point in \(V\), a point at infinity. This means the Voronoi cell is unbounded.

\text{voronoin} uses Qhull.

\[
[V,C] = \text{voronoin}(X, \text{options})
\] specifies a cell array of strings \(\text{options}\) to be used in Qhull. The default options are

- \{'Qbb'\} for 2- and 3-dimensional input
- \{'Qbb', 'Qx'\} for 4 and higher-dimensional input

If \(\text{options}\) is [], the default options are used. If \(\text{code}\) is {''}, no options are used, not even the default. For more information on Qhull and its options, see http://www.qhull.org.

**Visualization**

You can plot individual bounded cells of an \(n\)-dimensional Voronoi diagram. To do this, use \text{convhulln} to compute the vertices of the facets that make up the Voronoi cell. Then use \text{patch} and other plot functions to generate the figure.

**Examples**

**Example 1**

Let

\[
x = [\begin{array}{c} 0.5 \\ 0 \end{array}]
\]
then

\[ [V, C] = \text{voronoin}(x) \]

\[
V =
\begin{bmatrix}
0 & 0.5 \\
-0.5 & -0.5 \\
-0.2 & -0.1 \\
-0.1 & 0.1 \\
0.1 & -0.1 \\
0.1 & 0.1 \\
\end{bmatrix}
\]

\[
C =
\begin{bmatrix}
[1x4 \text{ double}] \\
[1x5 \text{ double}] \\
[1x4 \text{ double}] \\
[1x4 \text{ double}] \\
[1x4 \text{ double}] \\
[1x5 \text{ double}] \\
[1x4 \text{ double}] \\
\end{bmatrix}
\]

Use a for loop to see the contents of the cell array \( C \).

\[
\text{for } i=1:\text{length}(C), \text{ disp}(C{i}), \text{ end}
\]

\[
\begin{bmatrix}
4 & 2 & 1 & 3 \\
10 & 5 & 2 & 1 & 9 \\
\end{bmatrix}
\]
In particular, the fifth Voronoi cell consists of 4 points: \( V(10,:) \), \( V(5,:) \), \( V(6,:) \), \( V(8,:) \).

**Example 2**

The following example illustrates the options input to `voronoin`. The commands

\[
X = [-1 -1; 1 -1; 1 1; -1 1];
[V,C] = voronoin(X)
\]

return an error message.

? qhull input error: can not scale last coordinate. Input is cocircular
   or cospherical. Use option 'Qz' to add a point at infinity.

The error message indicates that you should add the option 'Qz'. The following command passes the option 'Qz', along with the default 'Qbb', to `voronoin`.

\[
[V,C] = voronoin(X,{'Qbb','Qz'})
V =
\]

\[
\begin{array}{cc}
\text{Inf} & \text{Inf} \\
0 & 0 \\
\end{array}
\]

\[
C =
\begin{bmatrix}
[1x2 \text{ double}] \\
[1x2 \text{ double}] \\
[1x2 \text{ double}] \\
\end{bmatrix}
\]
Algorithm

voronoin is based on Qhull [1]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also

convhull, convhulln, delaunay, delaunayn, voronoi

Reference

Purpose

Wait until timer stops running

Syntax

wait(obj)

Description

wait(obj) blocks the MATLAB command line and waits until the timer, represented by the timer object obj, stops running. When a timer stops running, the value of the timer object's Running property changes from 'on' to 'off'.

If obj is an array of timer objects, wait blocks the MATLAB command line until all the timers have stopped running.

If the timer is not running, wait returns immediately.

See Also

timer, start, stop
**Purpose**
Open or update a wait bar dialog box

**Syntax**

```matlab
h = waitbar(x, 'message')
waitbar(x, 'message', 'CreateCancelBtn', 'button_callback')
waitbar(x, 'message', property_name, property_value, ...)
waitbar(x)
waitbar(x, h)
waitbar(x, h, 'updated message')
```

**Description**
A wait bar is a figure that displays what percentage of a calculation is complete as the calculation proceeds by progressively filling a bar with red from left to right.

$h = \text{waitbar}(x, 'message')$ displays a wait bar of fractional length $x$. The wait bar figure displays until the code that controls it closes it or the user clicks its Close Window button. Its (figure) handle is returned in $h$. The argument $x$ must be between 0 and 1.

**Note** Wait bars are not modal figures (their WindowStyle is 'normal'). They often appear to be modal because the computational loops within which they are called prevent interaction with the Command Window until they terminate. For more information, see WindowStyle in the MATLAB Figure Properties documentation.

```matlab
waitbar(x, 'message', 'CreateCancelBtn', 'button_callback')
```

specifying CreateCancelBtn adds a Cancel button to the figure that executes the MATLAB commands specified in button_callback when the user clicks the Cancel button or the Close Figure button. `waitbar` sets both the Cancel button callback and the figure CloseRequestFcn to the string specified in button_callback.

```matlab
waitbar(x, 'message', property_name, property_value, ...)
```

optional arguments property_name and property_value enable you to set figure properties for the `waitbar`.
waitbar(x) subsequent calls to waitbar(x) extend the length of the bar to the new position x. Successive values of x normally increase. If they decrease, the wait bar runs in reverse.

waitbar(x,h) extends the length of the bar in the wait bar h to the new position x.

waitbar(x,h,'updated message') updates the message text in the waitbar figure, in addition to setting the fractional length to x.

Examples

Example 1 — Basic Wait Bar

Typically, you call waitbar repeatedly inside a for loop that performs a lengthy computation. For example:

```matlab
h = waitbar(0,'Please wait...');
steps = 1000;
for step = 1:steps
    % computations take place here
    waitbar(step / steps)
end
close(h)
```

Example 2 — Wait Bar with Dynamic Text and Cancel Button

Adding a Cancel button allows user to abort the computation. Clicking it sets a logical flag in the figure’s application data (appdata). The function tests for that value within the main loop and exits the loop as soon as the flag has been set. The example iteratively approximates the value of π. At each step, the current value is encoded as a string and displayed in the wait bar’s message field. When the function finishes,
it destroys the wait bar and returns the current estimate of \( \pi \) and the number of steps it ran.

Copy the following function to an M-file and save it as `approxpi.m`. Execute it as follows, allowing it to run for 10,000 iterations.

```matlab
[estimated_pi steps] = approxpi(10000)
```

You can click **Cancel** or close the window to abort the computation and return the current estimate of \( \pi \).

```matlab
function [valueofpi step] = approxpi(steps)
% Converge on pi in steps iterations, displaying waitbar.
% User can click Cancel or close button to exit the loop.
% Ten thousand steps yields error of about 0.001 percent.

h = waitbar(0,'1','Name','Approximating pi...','...CreateCancelButton',...setappdata(gcf,'canceling',1));
setappdata(h,'canceling',0)
% Approximate as \( \pi^2/8 = 1 + 1/9 + 1/25 + 1/49 + \ldots \)
pisqover8 = 1;
denom = 3;
valueofpi = sqrt(8 * pisqover8);
for step = 1:steps
    % Check for Cancel button press
    if getappdata(h,'canceling')
        break
    end
    % Report current estimate in the waitbar's message field
    waitbar(step/steps,h,sprintf('%12.9f',valueofpi))
    % Update the estimate
    pisqover8 = pisqover8 + 1 / (denom * denom);
denom = denom + 2;
valueofpi = sqrt(8 * pisqover8);
end
delete(h) % DELETE the waitbar; don't try to CLOSE it.
```
The function sets the figure Name property to describe what is being computed. In the for loop, calling waitbar sets the fractional progress indicator and displays intermediate results. The code `waitbar(i/steps,h,sprintf('%12.9f',valueofpi))` sets the wait bar’s message variable to a string representation of the current estimate of pi. Naturally, the extra computation involved makes iterations last longer than they need to, but such feedback can be helpful to users.

![Approximating pi](image)

**Note** You should call `delete` to remove a wait bar when you give it a `CloseRequestFcn`, as in the preceding code; calling `close` does not close it, and makes its Cancel and Close Window buttons unresponsive. This happens because the figure’s `CloseRequestFcn` recursively calls itself. In such a situation you must forcibly remove the wait bar, for example like this:

```matlab
set(0,'ShowHiddenHandles','on')
delete(get(0,'Children'))
```

However, as issuing these commands will delete all open figures—not just the wait bar—it is best never to use `close` in a `CloseRequestFcn` to close a window.

**See Also**

“Predefined Dialog Boxes” on page 1-110 for related functions

close, delete, dialog, msgbox, getappdata, setappdata
Purpose

Wait for condition before resuming execution

Syntax

waitfor(h)
waitfor(h,'PropertyName')
waitfor(h,'PropertyName',PropertyValue)

Description

The `waitfor` function blocks the caller's execution stream so that command-line expressions, callbacks, and statements in the blocked M-file do not execute until a specified condition is satisfied. It also blocks Simulink models from executing.

`waitfor(h)` returns when the graphics object identified by `h` is deleted or when a `Ctrl+C` is typed in the Command Window. If `h` does not exist, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName')`, in addition to the conditions in the previous syntax, returns when the value of `'PropertyName'` for the graphics object `h` changes. If `'PropertyName'` is not a valid property for the object, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName',PropertyValue)`, in addition to the conditions in the previous syntax, `waitfor` returns when the value of `'PropertyName'` for the graphics object `h` changes to `PropertyValue`. `waitfor` returns immediately without processing any events if `'PropertyName'` is set to `PropertyValue`.

Remarks

While `waitfor` blocks an execution stream, other execution streams in the form of callbacks may execute as a result of various events (e.g., pressing a mouse button). The functions `pause` (with no argument) and `uiwait` also block execution of MATLAB and Simulink in this manner.

`waitfor` can block nested execution streams. For example, a callback invoked during a `waitfor` statement can itself invoke `waitfor`.

See Also

keyboard, pause, uiresume, uiwait

“User Interface Development” on page 1-111 for related functions
**Purpose**
Wait for key press or mouse-button click

**Syntax**

```matlab
k = waitforbuttonpress
```

**Description**

`k = waitforbuttonpress` blocks the caller’s execution stream until the function detects that the user has clicked a mouse button or pressed a key while the figure window is active. The function returns

- 0 if it detects a mouse button click
- 1 if it detects a key press

Additional information about the event that causes execution to resume is available through the figure’s `CurrentCharacter`, `SelectionType`, and `CurrentPoint` properties.

If a `WindowButtonDownFcn` is defined for the figure, its callback is executed before `waitforbuttonpress` returns a value.

You can interrupt `waitforbuttonpress` by typing Ctrl+C, but an error results unless the function is called from within a try-catch block. You also receive an error from `waitforbuttonpress` if you close the figure by clicking the X close box unless you call `waitforbuttonpress` within a try-catch block.

**Example**

These statements display text in the Command Window when the user either clicks a mouse button or types a key in the figure window:

```matlab
w = waitforbuttonpress;
if w == 0
    disp('Button click')
else
    disp('Key press')
end
```

**See Also**

`dragrect`, `ginput`, `rbbox`, `waitfor`

“User Interface Development” on page 1-111 for related functions
**Purpose**
Open warning dialog box

**Syntax**

```matlab
h = warndlg
h = warndlg(warningstring)
h = warndlg(warningstring,dlgname)
h = warndlg(warningstring,dlgname,createmode)
```

**Description**

`h = warndlg` displays a dialog box named *Warning Dialog* containing the string *This is the default warning string*. The `warndlg` function returns the handle of the dialog box in `h`. The warning dialog box disappears after the user clicks **OK**.

`h = warndlg(warningstring)` displays a dialog box with the title *Warning Dialog* containing the string specified by `warningstring`. The `warningstring` argument can be any valid string format – cell arrays are preferred.

To use multiple lines in your warning, define `warningstring` using either of the following:

- `sprintf` with newline characters separating the lines
  ```matlab
  warndlg(sprintf('Message line 1 \n Message line 2'))
  ```

- Cell arrays of strings
  ```matlab
  warndlg({'Message line 1';'Message line 2'})
  ```

`h = warndlg(warningstring,dlgname)` displays a dialog box with title `dlgname`.

`h = warndlg(warningstring,dlgname,createmode)` specifies whether the warning dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `warningstring` and `dlgname`. The `createmode` argument can be a string or a structure.

If `createmode` is a string, it must be one of the values shown in the following table.
<table>
<thead>
<tr>
<th>createmode Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modal</td>
<td>Replaces the warning dialog box having the specified Title, that was last created or clicked on, with a modal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.</td>
</tr>
<tr>
<td>non-modal (default)</td>
<td>Creates a new nonmodal warning dialog box with the specified parameters. Existing warning dialog boxes with the same title are not deleted.</td>
</tr>
<tr>
<td>replace</td>
<td>Replaces the warning dialog box having the specified Title, that was last created or clicked on, with a nonmodal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.</td>
</tr>
</tbody>
</table>

**Note** A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

If you open a dialog with `errordlg`, `msgbox`, or `warndlg` using `'CreateMode','modal'` and a non-modal dialog created with any of these functions is already present and has the same name as the modal dialog, the non-modal dialog closes when the modal one opens.

For more information about modal dialog boxes, see `WindowStyle` in the Figure Properties.
If CreateMode is a structure, it can have fields WindowStyle and Interpreter. WindowStyle must be one of the options shown in the table above. Interpreter is one of the strings 'tex' or 'none'. The default value for Interpreter is 'none'.

**Examples**

The statement

```
warndlg('Pressing OK will clear memory','!! Warning !!')
```

displays this dialog box:

![Dialog Box](image)

**See Also**
dialog, errordlg, helpdlg, inputdlg, listdlg, msgbox, questdlg, figure, uistack, uitoolbar, warning

“Predefined Dialog Boxes” on page 1-110 for related functions
Purpose

Warning message

Syntax

warning('message')

warning('message', a1, a2,...)

warning('message_id', 'message')

warning('message_id', 'message', a1, a2, ..., an)

s = warning(state, 'message_id')

s = warning(state, mode)

Description

warning('message') displays descriptive text message and sets the warning state that lastwarn returns. If message is an empty string (''), warning resets the warning state but does not display any text.

warning('message', a1, a2,...) displays a message string that contains formatting conversion characters, such as those used with the MATLAB sprintf function. Each conversion character in message is converted to one of the values a1, a2, ... in the argument list.

Note MATLAB converts special characters (like \n and %d) in the warning message string only when you specify more than one input argument with warning. See Example 4 below.

warning('message_id', 'message') attaches a unique identifier, or message_id, to the warning message. The identifier enables you to single out certain warnings during the execution of your program, controlling what happens when the warnings are encountered. See “Message Identifiers” and “Warning Control” in the MATLAB Programming Fundamentals documentation for more information on the message_id argument and how to use it.

warning('message_id', 'message', a1, a2, ..., an) includes formatting conversion characters in message, and the character translations in arguments a1, a2, ..., an.

s = warning(state, 'message_id') is a warning control statement that enables you to indicate how you want MATLAB to act on certain
warning

warnings. The state argument can be 'on', 'off', or 'query'. The message_id argument can be a message identifier string, 'all', or 'last'. See “Warning Control Statements” in the MATLAB Programming Fundamentals documentation for more information.

Output s is a structure array that indicates the previous state of the selected warnings. The structure has the fields identifier and state. See “Output from Control Statements” in the MATLAB Programming Fundamentals documentation for more.

s = warning(state, mode) is a warning control statement that enables you to display an M-stack trace or display more information with each warning. The state argument can be 'on', 'off', or 'query'. The mode argument can be 'backtrace' or 'verbose'. See “Backtrace and Verbose Modes” in the MATLAB Programming Fundamentals documentation for more information.

Examples

**Example 1**

Generate a warning that displays a simple string:

```matlab
if ~ischar(p1)
    warning('Input must be a string')
end
```

**Example 2**

Generate a warning string that is defined at run-time. The first argument defines a message identifier for this warning:

```matlab
warning('MATLAB:paramAmbiguous', ...
    'Ambiguous parameter name, "%s".', param)
```

**Example 3**

Using a message identifier, enable just the actionNot Taken warning from Simulink by first turning off all warnings and then setting just that warning to on:

```matlab
warning off all
warning on Simulink:actionNotTaken
```
Use query to determine the current state of all warnings. It reports that you have set all warnings to off with the exception of Simulink:actionNotTaken:

```
warning query all
The default warning state is 'off'. Warnings not set to the default are
```

<table>
<thead>
<tr>
<th>State</th>
<th>Warning Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Simulink:actionNotTaken</td>
</tr>
</tbody>
</table>

**Example 4**

MATLAB converts special characters (like \n and %d) in the warning message string only when you specify more than one input argument with `warning`. In the single argument case shown below, \n is taken to mean backslash-n. It is not converted to a newline character:

```
warning('In this case, the newline \n is not converted.')
Warning: In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This is true regardless of whether the additional argument supplies conversion values or is a message identifier:

```
warning('WarnTests:convertTest', 'In this case, the newline \n is converted.')
Warning: In this case, the newline is converted.
```

**Example 5**

Turn on one particular warning, saving the previous state of this one warning in `s`. Remember that this nonquery syntax performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

After doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:
warning

warning(s)

See Also

lastwarn, warndlg, error, lasterror, errordlg, dbstop, disp, sprintf
Purpose

Waterfall plot

GUI Alternatives

To graph selected variables, use the Plot Selector in the Workspace Browser, or use the Figure Palette Plot Catalog. Manipulate graphs in plot edit mode with the Property Editor. For details, see Plotting Tools — Interactive Plotting in the MATLAB Graphics documentation and Creating Graphics from the Workspace Browser in the MATLAB Desktop Tools documentation.

Syntax

waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)
waterfall(axes_handles,...)
h = waterfall(...)

Description

The waterfall function draws a mesh similar to the meshz function, but it does not generate lines from the columns of the matrices. This produces a “waterfall” effect.

waterfall(Z) creates a waterfall plot using \( x = 1: \text{size}(Z,1) \) and \( y = 1: \text{size}(Z,1) \). \( Z \) determines the color, so color is proportional to surface height.

waterfall(X,Y,Z) creates a waterfall plot using the values specified in \( X, Y, \) and \( Z \). \( Z \) also determines the color, so color is proportional to the surface height. If \( X \) and \( Y \) are vectors, \( X \) corresponds to the columns of \( Z \), and \( Y \) corresponds to the rows, where \( \text{length}(x) = n, \text{length}(y) = m, \) and \( [m,n] = \text{size}(Z) \). \( X \) and \( Y \) are vectors or matrices that define the \( x \)- and \( y \)-coordinates of the plot. \( Z \) is a matrix that defines the \( z \)-coordinates of the plot (i.e., height above a plane). If \( C \) is omitted, color is proportional to \( Z \).

waterfall(...,C) uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of \( C \), which
must be the same size as Z. MATLAB performs a linear transformation on C to obtain colors from the current colormap.

`waterfall(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (gca).

`h = waterfall(...)` returns the handle of the patch graphics object used to draw the plot.

**Remarks**  
For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

**Examples**  
Produce a waterfall plot of the `peaks` function.

```matlab
[X,Y,Z] = peaks(30);
waterfall(X,Y,Z)
```
Algorithm

The range of X, Y, and Z, or the current setting of the axes Llim, YLim, and ZLim properties, determines the range of the axes (also set by axis). The range of C, or the current setting of the axes CLimit property, determines the color scaling (also set by caxis).

The CData property for the patch graphics objects specifies the color at every point along the edge of the patch, which determines the color of the lines.

The waterfall plot looks like a mesh surface; however, it is a patch graphics object. To create a surface plot similar to waterfall, use the meshz function and set the MeshStyle property of the surface to 'Row'.

2-4153
For a discussion of parametric surfaces and related color properties, see \texttt{surf}.

\textbf{See Also}

\texttt{axes}, \texttt{axis}, \texttt{caxis}, \texttt{meshz}, \texttt{ribbon}, \texttt{surf}

Properties for patch graphics objects
Purpose  Information about Microsoft WAVE (.wav) sound file

Syntax  

[m d] = wavfinfo(filename)

Description  

[m d] = wavfinfo(filename) returns information about the contents of the WAVE sound file specified by the string filename. Enclose the filename input in single quotes.

m is the string 'Sound (WAV) file', if filename is a WAVE file. Otherwise, it contains an empty string ('').

d is a string that reports the number of samples in the file and the number of channels of audio data. If filename is not a WAVE file, it contains the string 'Not a WAVE file'.

See Also  wavplay, wavread, wavrecord, wavwrite
**wavplay**

**Purpose**
Play recorded sound on PC-based audio output device

**Syntax**

```matlab
wavplay(y,Fs)
wavplay(...,'mode')
```

**Description**

`wavplay(y,Fs)` plays the audio signal stored in the vector `y` on a PC-based audio output device. You specify the audio signal sampling rate with the integer `Fs` in samples per second. The default value for `Fs` is 11025 Hz (samples per second). `wavplay` supports only 1- or 2-channel (mono or stereo) audio signals.

`wavplay(...,'mode')` specifies how `wavplay` interacts with the command line, according to the string `'mode'`. The string `'mode'` can be

- `'async'`: You have immediate access to the command line as soon as the sound begins to play on the audio output device (a nonblocking device call).
- `'sync'` (default value): You don’t have access to the command line until the sound has finished playing (a blocking device call).

The audio signal `y` can be one of four data types. The number of bits used to quantize and play back each sample depends on the data type.

**Data Types for wavplay**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Quantization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-precision (default value)</td>
<td>16 bits/sample</td>
</tr>
<tr>
<td>Single-precision</td>
<td>16 bits/sample</td>
</tr>
<tr>
<td>16-bit signed integer</td>
<td>16 bits/sample</td>
</tr>
<tr>
<td>8-bit unsigned integer</td>
<td>8 bits/sample</td>
</tr>
</tbody>
</table>

**Remarks**
You can play your signal in stereo if `y` is a two-column matrix.
Examples

The MAT-files gong.mat and chirp.mat both contain an audio signal \( y \) and a sampling frequency \( F_s \). Load and play the gong and the chirp audio signals. Change the names of these signals in between `load` commands and play them sequentially using the `sync` option for `wavplay`.

```matlab
load chirp;
y1 = y; Fs1 = Fs;
load gong;
wavplay(y1,Fs1,'sync') % The chirp signal finishes before the
wavplay(y,Fs) % gong signal begins playing.
```

See Also

`wavfinfo`, `wavread`, `wavrecord`, `wavwrite`
wavread

**Purpose**
Read Microsoft WAVE (.wav) sound file

**Graphical Interface**
As an alternative to wavread, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

**Syntax**

- `y = wavread(filename)`
- `[y, Fs, nbits] = wavread(filename)`
- `[..., N] = wavread(filename, N)`
- `[..., [N1 N2]] = wavread(filename)`
- `y = wavread(filename, fmt)`
- `siz = wavread(filename, 'size')`
- `[..., y, fs, nbits, opts] = wavread(...)`

**Description**

`y = wavread(filename)` loads a WAVE file specified by `filename`, returning the sampled data in `y`. The `filename` input is a string enclosed in single quotes. The .wav extension is appended if no extension is given.

`[y, Fs, nbits] = wavread(filename)` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`nbits`) used to encode the data in the file.

`[..., N] = wavread(filename, N)` returns only the first `N` samples from each channel in the file.

`[..., [N1 N2]] = wavread(filename)` returns only samples `N1` through `N2` from each channel in the file.

`y = wavread(filename, fmt)` specifies the data type format of `y` used to represent samples read from the file. `fmt` can be either of the following values.
### Value Description

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'double'</td>
<td>y contains double-precision normalized samples. This is the default value, if <code>fmt</code> is omitted.</td>
</tr>
<tr>
<td>'native'</td>
<td>y contains samples in the native data type found in the file. Interpretation of <code>fmt</code> is case-insensitive, and partial matching is supported.</td>
</tr>
</tbody>
</table>

`siz = wavread(filename,'size')` returns the size of the audio data contained in `filename` in place of the actual audio data, returning the vector `siz = [samples channels]`.

`[y, fs, nbits, opts] = wavread(...)` returns a structure `opts` of additional information contained in the WAV file. The content of this structure differs from file to file. Typical structure fields include `opts.fmt` (audio format information) and `opts.info` (text which may describe title, author, etc.).

### Output Scaling

The range of values in `y` depends on the data format `fmt` specified. Some examples of output scaling based on typical bit-widths found in a WAV file are given below for both 'double' and 'native' formats.

#### Native Formats

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>MATLAB Data Type</th>
<th>Data Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>uint8 (unsigned integer)</td>
<td>0 &lt;= y &lt;= 255</td>
</tr>
<tr>
<td>16</td>
<td>int16 (signed integer)</td>
<td>-32768 &lt;= y &lt;= +32767</td>
</tr>
<tr>
<td>24</td>
<td>int32 (signed integer)</td>
<td>-2^23 &lt;= y &lt;= +2^23-1</td>
</tr>
<tr>
<td>32</td>
<td>single (floating point)</td>
<td>-1.0 &lt;= y &lt; +1.0</td>
</tr>
</tbody>
</table>
Double Formats

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>MATLAB Data Type</th>
<th>Data Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>N&lt;32</td>
<td>double</td>
<td>-1.0 &lt;= y &lt; +1.0</td>
</tr>
<tr>
<td>N=32</td>
<td>double</td>
<td>-1.0 &lt;= y &lt;= +1.0</td>
</tr>
</tbody>
</table>

Note: Values in y might exceed -1.0 or +1.0 for the case of N=32 bit data samples stored in the WAV file.

wavread supports multi-channel data, with up to 32 bits per sample.
wavread supports Pulse-code Modulation (PCM) data format only.

See Also
auread, auwrite, wavinfo, wavplay, wavrecord, wavwrite
**Purpose**
Record sound using PC-based audio input device

**Syntax**

```matlab
y = wavrecord(n,Fs)
y = wavrecord(...,ch)
y = wavrecord(...,'dtype')
```

**Description**

- `y = wavrecord(n,Fs)` records `n` samples of an audio signal, sampled at a rate of `Fs` Hz (samples per second). The default value for `Fs` is 11025 Hz.
- `y = wavrecord(...,ch)` uses `ch` number of input channels from the audio device. `ch` can be either 1 or 2, for mono or stereo, respectively. The default value for `ch` is 1.
- `y = wavrecord(...,'dtype')` uses the data type specified by the string `'dtype'` to record the sound. The following table lists the string values for `'dtype'` along with the corresponding bits per sample and acceptable data range for `y`.

<table>
<thead>
<tr>
<th>dtype</th>
<th>Bits/sample</th>
<th>y Data Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>'double'</td>
<td>16</td>
<td>-1.0 &lt;= y &lt;= +1.0</td>
</tr>
<tr>
<td>'single'</td>
<td>16</td>
<td>-1.0 &lt;= y &lt;= +1.0</td>
</tr>
<tr>
<td>'int16'</td>
<td>16</td>
<td>-32768 &lt;= y &lt;= +32767</td>
</tr>
<tr>
<td>'uint8'</td>
<td>8</td>
<td>0 &lt;= y &lt;= 255</td>
</tr>
</tbody>
</table>

**Remarks**
Standard sampling rates for PC-based audio hardware are 8000, 11025, 2250, and 44100 samples per second. Stereo signals are returned as two-column matrices. The first column of a stereo audio matrix corresponds to the left input channel, while the second column corresponds to the right input channel.

The `wavrecord` function is for use only with 32-bit Microsoft Windows operating systems. To record audio data from audio input devices on other platforms, use `audiorecorder`. 
wavrecord

Examples
Record 5 seconds of 16-bit audio sampled at 11025 Hz. Play back the recorded sound using wavplay. Speak into your audio device (or produce your audio signal) while the wavrecord command runs.

\[
Fs = 11025;
\]
\[
y = \text{wavrecord}(5*Fs,Fs,'\text{int16}');
\]
\[
\text{wavplay}(y,Fs);
\]

See Also
audiorecorder, wavinfo, wavplay, wavread, wavwrite
Purpose
Write Microsoft WAVE (.wav) sound file

Syntax
wavwrite(y,filename)
wavwrite(y,Fs,filename)
wavwrite(y,Fs,N,filename)

Description
wavwrite(y,filename) writes the data stored in the variable y to a
WAVE file called filename. The filename input is a string enclosed in
single quotes. The data has a sample rate of 8000 Hz and is assumed
to be 16-bit. Each column of the data represents a separate channel.
Therefore, stereo data should be specified as a matrix with two columns.

wavwrite(y,Fs,filename) writes the data stored in the variable y to
a WAVE file called filename. The data has a sample rate of Fs Hz
and is assumed to be 16-bit.

wavwrite(y,Fs,N,filename) writes the data stored in the variable y to
a WAVE file called filename. The data has a sample rate of Fs Hz and
is N-bit, where N is 8, 16, 24, or 32.

Input Data Ranges
The range of values in y depends on the number of bits specified by N
and the data type of y. The following tables list the valid input ranges
based on the value of N and the data type of y.

If y contains integer data:

<table>
<thead>
<tr>
<th>N Bits</th>
<th>y Data Type</th>
<th>y Data Range</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>uint8</td>
<td>0 &lt;= y &lt;= 255</td>
<td>uint8</td>
</tr>
<tr>
<td>16</td>
<td>int16</td>
<td>-32768 &lt;= y &lt;= +32767</td>
<td>int16</td>
</tr>
<tr>
<td>24</td>
<td>int32</td>
<td>-2^23 &lt;= y &lt;= 2^23 – 1</td>
<td>int32</td>
</tr>
</tbody>
</table>
If \( y \) contains floating-point data:

<table>
<thead>
<tr>
<th>N Bits</th>
<th>( y ) Data Type</th>
<th>( y ) Data Range</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>single or double</td>
<td>(-1.0 \leq y &lt; +1.0)</td>
<td>uint8</td>
</tr>
<tr>
<td>16</td>
<td>single or double</td>
<td>(-1.0 \leq y &lt; +1.0)</td>
<td>int16</td>
</tr>
<tr>
<td>24</td>
<td>single or double</td>
<td>(-1.0 \leq y &lt; +1.0)</td>
<td>int32</td>
</tr>
<tr>
<td>32</td>
<td>single or double</td>
<td>(-1.0 \leq y \leq +1.0)</td>
<td>single</td>
</tr>
</tbody>
</table>

For floating point data where \( N < 32 \), amplitude values are clipped to the range \(-1.0 \leq y < +1.0\).

**Note** 8-, 16-, and 24-bit files are type 1 integer pulse code modulation (PCM). 32-bit files are written as type 3 normalized floating point.

**See Also**

auwrite, wavinfo, wavplay, wavread, wavrecord
Purpose
Day of week

Syntax
[N, S] = weekday(D)
[N, S] = weekday(D, form)
[N, S] = weekday(D, locale)
[N, S] = weekday(D, form, locale)

Description
[N, S] = weekday(D) returns the day of the week in numeric (N) and string (S) form for a given serial date number or date string D. Input argument D can represent more than one date in an array of serial date numbers or a cell array of date strings.

[N, S] = weekday(D, form) returns the day of the week in numeric (N) and string (S) form, where the content of S depends on the form argument. If form is 'long', then S contains the full name of the weekday (e.g., Tuesday). If form is 'short', then S contains an abbreviated name (e.g., Tues) from this table.

The days of the week are assigned these numbers and abbreviations.

<table>
<thead>
<tr>
<th>N</th>
<th>S (short)</th>
<th>S (long)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sun</td>
<td>Sunday</td>
</tr>
<tr>
<td>2</td>
<td>Mon</td>
<td>Monday</td>
</tr>
<tr>
<td>3</td>
<td>Tue</td>
<td>Tuesday</td>
</tr>
<tr>
<td>4</td>
<td>Wed</td>
<td>Wednesday</td>
</tr>
<tr>
<td>5</td>
<td>Thu</td>
<td>Thursday</td>
</tr>
<tr>
<td>6</td>
<td>Fri</td>
<td>Friday</td>
</tr>
<tr>
<td>7</td>
<td>Sat</td>
<td>Saturday</td>
</tr>
</tbody>
</table>

[N, S] = weekday(D, locale) returns the day of the week in numeric (N) and string (S) form, where the format of the output depends on the locale argument. If locale is 'local', then weekday uses local format for its output. If locale is 'en_US', then weekday uses US English.
weekday

[N, S] = weekday(D, form, locale) returns the day of the week using the formats described above for form and locale.

Examples

Either

    [n, s] = weekday(728647)

or

    [n, s] = weekday('19-Dec-1994')

returns n = 2 and s = Mon.

See Also
datenum, datevec, eomday
Purpose
List MATLAB files in current directory

Graphical Interface
As an alternative to the `what` function, use the Current Directory browser.

Syntax

```matlab
what
gwhat dirname
gwhat classname
gwhat packagename

s = what('dirname')
```

Description
what lists the path for the current working directory, and all files and directories related to MATLAB found in the working directory. Files listed are M, MAT, MEX, MDL, and P-files. Directories listed are all class and package directories.

what `dirname` lists path, file, and directory information for directory `dirname`. If this directory is on the MATLAB search path, it is not necessary to enter the full path to `dirname`. The last one or two components of the path are sufficient.

what `classname` lists path, file, and directory information for method directory `@classname`. For example, what `cfit` lists the MATLAB files and directories in `toolbox/curvefit/curvefit/@cfit`.

what `packagename` lists path, file, and directory information for package directory `+packagename`. For example, what `commsrc` lists the MATLAB files and directories in `toolbox/comm/comm/+commsrc`.

`s = what('dirname')` returns the results in a structure array with the fields shown in the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>Path to directory</td>
</tr>
<tr>
<td>m</td>
<td>Cell array of M-file names</td>
</tr>
<tr>
<td>mat</td>
<td>Cell array of MAT-file names</td>
</tr>
</tbody>
</table>
what

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mex</td>
<td>Cell array of MEX-file names</td>
</tr>
<tr>
<td>mdl</td>
<td>Cell array of MDL-file names</td>
</tr>
<tr>
<td>p</td>
<td>Cell array of P-file names</td>
</tr>
<tr>
<td>classes</td>
<td>Cell array of class directories</td>
</tr>
<tr>
<td>packages</td>
<td>Cell array of package directories</td>
</tr>
</tbody>
</table>

Examples

List MATLAB Files in a Directory

List the files in toolbox/matlab/audiovideo:

```matlab
what audiovideo
```

M-files in directory matlabroot\toolbox\matlab\audiovideo

- avifinfo
- aviinfo
- sound
- soundsc
- aviread
- wavfinfo
- lin2mu
- wavplay
- mmcompinfo
- wavread
- mmfileinfo
- wavrecord
- movie2avi
- wavwrite
- mu2lin
- prefpanel

MAT-files in directory matlabroot\toolbox\matlab\audiovideo

- chirp
- handel
- splat
- gong
- laughter
- train

MEX-files in directory matlabroot\toolbox\matlab\audiovideo

- winaudioplayer
- winaudiorecorder
Classes in directory matlabroot\toolbox\matlab\audiovideo

audioplayer    avifile
audiorecorder  mmreader

Return File Names to Structure

Obtain a structure array containing the MATLAB file names in toolbox/matlab/general:

```matlab
s = what('general')
s =
    path: 'matlabroot:\toolbox\matlab\general'
        m: {89x1 cell}
        mat: {0x1 cell}
        mex: {2x1 cell}
        mdl: {0x1 cell}
            p: {'callgraphviz.p'}
        classes: {'char'}
        packages: {0x1 cell}
```

List M-Files in a Package

Find the supporting M-files for one of the packages in the Communications Toolbox product:

```matlab
p1 = what('comm');
p1.packages
ans =
    'commdevice'
    'crc'
    'commsrc'

p2 = what('commsrc');
p2.m
ans =
    'abstractJitter.m'
```
what

'abstractPulse.m'
'combinedjitter.m'
'diracjitter.m'
'periodicjitter.m'
'randomjitter.m'

See Also

dir, exist, lookfor, mfilename, path, which, who

“Managing Files and Working with the Current Directory”
**Purpose**
Release Notes for MathWorks products

**Syntax**
whatsnew

**Description**
whatsnew displays the Release Notes in the Help browser, presenting information about new features, problems from previous releases that have been fixed in the current release, and compatibility issues, all organized by product.

**See Also**
help, version
**Purpose**

Locate functions and files

**Graphical Interface**

As an alternative to the `which` function, you can use the Current Directory browser to find files. You can find functions using the Function Browser in the Command Window or Editor.

**Syntax**

```
which fun
which classname/fun
which private/fun
which classname/private/fun
which fun1 in fun2
which fun(a,b,c,...)
which file.ext
which fun -all
s = which('fun',...)
```

**Description**

`which fun` displays the full pathname for the argument `fun`. If `fun` is a

- MATLAB function or Simulink model in an M, P, or MDL file on the MATLAB path, then `which` displays the full pathname for the corresponding file
- Workspace variable, then `which` displays a message identifying `fun` as a variable
- Method in a loaded Java class, then `which` displays the package, class, and method name for that method

If `fun` is an overloaded function or method, then `which fun` returns only the pathname of the first function or method found.

`which classname/fun` displays the full pathname for the M-file defining the `fun` method in MATLAB class, `classname`. For example, `which serial/fopen` displays the path for `fopen.m` in the MATLAB class directory, `@serial`. 2-4172
which `private`/fun limits the search to private functions. For example, which `private/orthog` displays the path for `orthog.m` in the `/private subdirectory of toolbox/matlab/elmat`.

which `classname/private`/fun limits the search to private methods defined by the MATLAB class, `classname`. For example, which `dfilt/private/todtf` displays the path for `todtf.m` in the private directory of the `dfilt` class.

which `fun1 in fun2` displays the pathname to function `fun1` in the context of the M-file `fun2`. You can use this form to determine whether a subfunction is being called instead of a function on the path. For example, which `get in editpath` tells you which `get` function is called by `editpath.m`.

During debugging of `fun2`, using which `fun1` gives the same result.

which `fun(a,b,c,...)` displays the path to the specified function with the given input arguments. For example, which `feval(g)`, when `g=inline('sin(x)')`, indicates that `inline/feval.m` would be invoked. which `toLowerCase(s)`, when `s=java.lang.String('my Java string')`, indicates that the `toLowerCase` method in class `java.lang.String` would be invoked.

which `file.ext` displays the full pathname of the specified file if that file is in the current working directory or on the MATLAB path. To display the path for a file that has no file extension, type “which file.” (the period following the filename is required). Use `exist` to check for the existence of files anywhere else.

which `fun -all` displays the paths to all items on the MATLAB path with the name `fun`. You may use the `-all` qualifier with any of the above formats of the which function.

`s = which('fun',...)` returns the results of which in the string `s`. For workspace variables, `s` is the string 'variable'. You may specify an output variable in any of the above formats of the which function.

If `-all` is used with this form, the output `s` is always a cell array of strings, even if only one string is returned.
Examples

The statement below indicates that `pinv` is in the `matfun` directory of MATLAB.

```matlab
which pinv
matlabroot\toolbox\matlab\matfun\pinv.m
```

To find the `fopen` function used on MATLAB `serial` class objects

```matlab
which serial/fopen
matlabroot\toolbox\matlab\iofun\@serial\fopen.m % serial method
```

To find the `setMonth` method used on objects of the Java `Date` class, the class must first be loaded into MATLAB. The class is loaded when you create an instance of the class:

```matlab
myDate = java.util.Date;
which setMonth
```

MATLAB displays:

```matlab
setMonth is a Java method % java.util.Date method
```

When you specify an output variable, which returns a cell array of strings to the variable. You must use the `function` form of `which`, enclosing all arguments in parentheses and single quotes:

```matlab
s = which('private/stradd','-all');
whos s
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>3x1</td>
<td>562</td>
<td>cell array</td>
</tr>
</tbody>
</table>

Grand total is 146 elements using 562 bytes

See Also

dir, doc, exist, lookfor, mfilename, path, type, what, who
**Purpose**
Repeatedly execute statements while condition is true

**Syntax**
while expression, statements, end

**Description**
while expression, statements, end repeatedly executes one or more MATLAB statements in a loop, continuing until expression no longer holds true or until MATLAB encounters a break, or return instruction. thus forcing an immediately exit of the loop. If MATLAB encounters a continue statement in the loop code, it immediately exits the current pass at the location of the continue statement, skipping any remaining code in that pass, and begins another pass at the start of the loop statements.

expression is a MATLAB expression that evaluates to a result of logical 1 (true) or logical 0 (false). expression can be scalar or an array. It must contain all real elements, and the statement all(A(:)) must be equal to logical 1 for the expression to be true.

expression usually consists of variables or smaller expressions joined by relational operators (e.g., count < limit) or logical functions (e.g., isreal(A)). Simple expressions can be combined by logical operators (&&, ||, ~) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to “Operator Precedence” rules.

(count < limit) && ((height - offset) >= 0)

statements is one or more MATLAB statements to be executed only while the expression is true or nonzero.

The scope of a while statement is always terminated with a matching end.

See “Program Control Statements” in the MATLAB Programming Fundamentals documentation for more information on controlling the flow of your program code.
Remarks

Nonscalar Expressions

If the evaluated expression yields a nonscalar value, then every element of this value must be true or nonzero for the entire expression to be considered true. For example, the statement while \((A < B)\) is true only if each element of matrix \(A\) is less than its corresponding element in matrix \(B\). See “Example 2 – Nonscalar Expression” on page 2-4177, below.

Partial Evaluation of the Expression Argument

Within the context of an if or while expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is true or false through only partial evaluation.

For example, if \(A\) equals zero in statement 1 below, then the expression evaluates to false, regardless of the value of \(B\). In this case, there is no need to evaluate \(B\) and MATLAB does not do so. In statement 2, if \(A\) is nonzero, then the expression is true, regardless of \(B\). Again, MATLAB does not evaluate the latter part of the expression.

1) while \((A && B)\)  
2) while \((A || B)\)

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

\[
\text{while } (b \neq 0) && (a/b > 18.5) \\
\text{if exist('myfun.m')} && (myfun(x) \geq y) \\
\text{if iscell(A)} && \text{all(cellfun('isreal', A))}
\]

Empty Arrays

In most cases, using while on an empty array returns false. There are some conditions however under which while evaluates as true on an empty array. Two examples of this are

\[
\text{A} = []; \\
\text{while all(A), do_something, end} \\
\text{while 1|A, do_something, end}
\]
Short-Circuiting Behavior

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

See “Short-Circuiting in Elementwise Operators” for more information on this.

Examples

Example 1 – Simple while Statement

The variable eps is a tolerance used to determine such things as near singularity and rank. Its initial value is the machine epsilon, the distance from 1.0 to the next largest floating-point number on your machine. Its calculation demonstrates while loops.

```matlab
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
eps = eps*2
```

This example is for the purposes of illustrating while loops only and should not be executed in your MATLAB session. Doing so will disable the eps function from working in that session.

Example 2 – Nonscalar Expression

Given matrices A and B,

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>1 1</td>
</tr>
<tr>
<td>2 3</td>
<td>3 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates As</th>
<th>Because</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &lt; B</td>
<td>false</td>
<td>A(1,1) is not less than B(1,1).</td>
</tr>
</tbody>
</table>
### while

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates As</th>
<th>Because</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &lt; (B + 1)</td>
<td>true</td>
<td>Every element of A is less than that same element of B with 1 added.</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>false</td>
<td>A(1,2) is false, and B is ignored due to short-circuiting.</td>
</tr>
<tr>
<td>B &lt; 5</td>
<td>true</td>
<td>Every element of B is less than 5.</td>
</tr>
</tbody>
</table>

**See Also**
end, for, break, continue, return, all, any, if, switch
Purpose
Change axes background color

Syntax
whitebg
whitebg(fig)
whitebg(ColorSpec)
whitebg(fig, ColorSpec)

Description
whitebg complements the colors in the current figure.
whitebg(fig) complements colors in all figures specified in the vector
fig.
whitebg(ColorSpec) and whitebg(fig, ColorSpec) change the
color of the axes, which are children of the figure, to the color
specified by ColorSpec. Without a figure specification, whitebg or
whitebg(ColorSpec) affects the current figure and the root's default
properties so subsequent plots and new figures use the new colors.
whitebg(fig, ColorSpec) sets the default axes background color of
the figures in the vector fig to the color specified by ColorSpec. Other
axes properties and the figure background color can change as well so
that graphs maintain adequate contrast. ColorSpec can be a 1-by-3
RGB color or a color string such as 'white' or 'w'.
whitebg(fig) complements the colors of the objects in the specified
figures. This syntax is typically used to toggle between black and white
axes background colors, and is where whitebg gets its name. Include
the root window handle (0) in fig to affect the default properties for
new windows or for clf reset.

Remarks
whitebg works best in cases where all the axes in the figure have the
same background color.
whitebg changes the colors of the figure's children, with the exception
of shaded surfaces. This ensures that all objects are visible against the
new background color. whitebg sets the default properties on the root
such that all subsequent figures use the new background color.
whitebg

**Examples**

Set the background color to blue-gray.

whitebg([0.5, 0.6])

Set the background color to blue.

whitebg('blue')

**See Also**

ColorSpec, colordef

The figure graphics object property InvertHardCopy

“Color Operations” on page 1-105 for related functions
**Purpose**

List variables in workspace

**Graphical Interface**

As an alternative to `whos`, use the Workspace browser. Or use the Current Directory browser to view the contents of MAT-files without loading them.

**Syntax**

```
who
whos
who(variable_list)
whos(variable_list)
who(variable_list, qualifiers)
whos(variable_list, qualifiers)
s = who(variable_list, qualifiers)
s = whos(variable_list, qualifiers)
who variable_list qualifiers
whos variable_list qualifiers
```

Each of these syntaxes applies to both `who` and `whos`:

**Description**

`who` lists in alphabetical order all variables in the currently active workspace.

`whos` lists in alphabetical order all variables in the currently active workspace along with their sizes and types. It also reports the totals for sizes.

**Note** If `who` or `whos` is executed within a nested function, the MATLAB software lists the variables in the workspace of that function and in the workspaces of all functions containing that function. See the Remarks section, below.

`who(variable_list)` and `whos(variable_list)` list only those variables specified in `variable_list`, where `variable_list` is a comma-delimited list of quoted strings: `'var1'`, `'var2'`, ..., `'varN'`. You can use the wildcard character `*` to display variables that
match a pattern. For example, `who('A*')` finds all variables in the current workspace that start with A.

`who(variable_list, qualifiers)` and `whos(variable_list, qualifiers)` list those variables in `variable_list` that meet all qualifications specified in `qualifiers`. You can specify any or all of the following qualifiers, and in any order.

<table>
<thead>
<tr>
<th>Qualifier Syntax</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>'global'</td>
<td>List variables in the global workspace.</td>
<td><code>whos('global')</code></td>
</tr>
<tr>
<td>'-file', filename</td>
<td>List variables in the specified MAT-file. Use the full path for filename.</td>
<td><code>whos('-file', 'mydata')</code></td>
</tr>
<tr>
<td>'-regexp', exprlist</td>
<td>List variables that match any of the regular expressions in exprlist.</td>
<td><code>whos('-regexp', '[AB].', '\w\d')</code></td>
</tr>
</tbody>
</table>

`s = who(variable_list, qualifiers)` returns cell array `s` containing the names of the variables specified in `variable_list` that meet the conditions specified in `qualifiers`.

`s = whos(variable_list, qualifiers)` returns structure `s` containing the following fields for the variables specified in `variable_list` that meet the conditions specified in `qualifiers`:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the variable</td>
</tr>
<tr>
<td>size</td>
<td>Dimensions of the variable array</td>
</tr>
<tr>
<td>bytes</td>
<td>Number of bytes allocated for the variable array</td>
</tr>
<tr>
<td>class</td>
<td>Class of the variable. Set to the string '(unassigned)' if the variable has no value.</td>
</tr>
</tbody>
</table>
who variable_list qualifiers and whos variable_list qualifiers are the unquoted forms of the syntax. Both variable_list and qualifiers are space-delimited lists of unquoted strings.

**Remarks**

**Nested Functions.** When you use who or whos inside of a nested function, MATLAB returns or displays all variables in the workspace of that function, and in the workspaces of all functions in which that function is nested. This applies whether you include calls to who or whos in your M-file code or if you call who or whos from the MATLAB debugger.

If your code assigns the output of whos to a variable, MATLAB returns the information in a structure array containing the fields described above. If you do not assign the output to a variable, MATLAB displays the information at the Command Window, grouped according to workspace.

If your code assigns the output of who to a variable, MATLAB returns the variable names in a cell array of strings. If you do not assign the output, MATLAB displays the variable names at the Command Window, but not grouped according to workspace.
**Compressed Data.** Information returned by the command `whos -file` is independent of whether the data in that file is compressed or not. The byte counts returned by this command represent the number of bytes data occupies in the MATLAB workspace, and not in the file the data was saved to. See the function reference for `save` for more information on data compression.

**MATLAB Objects.** `whos -file filename` does not return the sizes of any MATLAB objects that are stored in file `filename`.

**Examples**

**Example 1**

Show variable names starting with the letter `a`:

```matlab
who a*
```

Show variables stored in MAT-file `mydata.mat`:

```matlab
who -file mydata
```

**Example 2**

Return information on variables stored in file `mydata.mat` in structure array `s`:

```matlab
s = whos('-file', 'mydata1')
s =
6x1 struct array with fields:
    name
    size
    bytes
    class
    global
    sparse
    complex
    nesting
    persistent
```
Display the name, size, and class of each of the variables returned by whos:

```
for k=1:length(s)
    disp([' ' s(k).name ' ' mat2str(s(k).size) ' ' s(k).class])
end
```

- `A` [1 1] double
- `spArray` [5 5] double
- `strArray` [2 5] cell
- `x` [3 2 2] double
- `y` [4 5] cell

**Example 3**

Show variables that start with `java` and end with `Array`. Also show their dimensions and class name:

```
whos -file mydata2 -regexp <java.*Array>
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>javaChrArray</td>
<td>3x1</td>
<td></td>
<td>java.lang.String[][][]</td>
</tr>
<tr>
<td>javaDblArray</td>
<td>4x1</td>
<td></td>
<td>java.lang.Double[][][]</td>
</tr>
<tr>
<td>javaIntArray</td>
<td>14x1</td>
<td></td>
<td>java.lang.Integer[][][]</td>
</tr>
</tbody>
</table>

**Example 4**

The function shown here uses variables with persistent, global, sparse, and complex attributes:

```
function show_attributes
    persistent p;
    global g;
    o = 1;    g = 2;
    s = sparse(eye(5));
    c = [4+5i 9-3i 7+6i];
    whos

When the function is run, whos displays these attributes:

show_attributes
```
who, whos

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1x3</td>
<td>48</td>
<td>double</td>
<td>complex</td>
</tr>
<tr>
<td>g</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td>global</td>
</tr>
<tr>
<td>p</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td>persistent</td>
</tr>
<tr>
<td>s</td>
<td>5x5</td>
<td>84</td>
<td>double</td>
<td>sparse</td>
</tr>
</tbody>
</table>

**Example 5**

Function whos_demo contains two nested functions. One of these functions calls whos; the other calls who:

```matlab
function whos_demo
    date_time = datestr(now);
    [str pos] = textscan(date_time, '%s%s%s', ...
        1, 'delimiter', '- :');
    get_date(str);
    str = textscan(date_time(pos+1:end), '%s%s%s', ...
        1, 'delimiter', '- :');
    get_time(str);

    function get_date(d)
        day = d{1}; mon = d{2}; year = d{3};
        whos
    end
    function get_time(t)
        hour = t{1}; min = t{2}; sec = t{3};
        who
    end
end
```

When nested function get_date calls whos, MATLAB displays information on the variables in all workspaces that are in scope at the time. This includes nested function get_date and also the function in which it is nested, whos_demo. The information is grouped by workspace:
When nested function `get_time` calls `who`, MATLAB displays names of the variables in the workspaces that are in scope at the time. This includes nested function `get_time` and also the function in which it is nested, `whos_demo`. The information is not grouped by workspace in this case:

> Your variables are:

```
    hour min sec t ans  date_time
pos  str
```

**See Also**

- `assignin`, `clear`, `computer`, `dir`, `evalin`, `exist`, `inmem`, `load`, `save`, `what`, `workspace`

“MATLAB Workspace” in the Desktop Tools and Development Environment documentation
Purpose
Wilkinson’s eigenvalue test matrix

Syntax
W = wilkinson(n)

Description
W = wilkinson(n) returns one of J. H. Wilkinson’s eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Examples
wilkinson(7)

ans =

     3     1     0     0     0     0     0
     1     2     1     0     0     0     0
     0     1     1     1     0     0     0
     0     0     1     0     1     0     0
     0     0     0     1     1     1     0
     0     0     0     0     1     2     1
     0     0     0     0     0     1     3

The most frequently used case is wilkinson(21). Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

See Also
eig, gallery, pascal
Purpose
Open file in appropriate application (Windows)

Syntax
winopen(filename)

Description
winopen(filename) opens filename in the appropriate Microsoft Windows application. The filename input is a string enclosed in single quotes. The winopen function uses the appropriate Windows shell command, and performs the same action as if you double-click the file in the Windows Explorer program. If filename is not in the current directory, specify the absolute path for filename.

Examples
Open the file thesis.doc, located in the current directory, in the Microsoft Word program:

    winopen('thesis.doc')

Open myresults.html in your system Web browser:

    winopen('D:/myfiles/myresults.html')

See Also
dos, open, web
**Purpose**

Item from Windows registry

**Syntax**

```matlab
valnames = winqueryreg('name', 'rootkey', 'subkey')
value = winqueryreg('rootkey', 'subkey', 'valname')
value = winqueryreg('rootkey', 'subkey')
```

**Description**

`valnames = winqueryreg('name', 'rootkey', 'subkey')` returns all value names in `rootkey\subkey` of Microsoft Windows operating system registry to a cell array of strings. The first argument is the literal quoted string, `'name'`.

`value = winqueryreg('rootkey', 'subkey', 'valname')` returns the value for value name `valname` in `rootkey\subkey`.

If the value retrieved from the registry is a string, `winqueryreg` returns a string. If the value is a 32-bit integer, `winqueryreg` returns the value as an integer of the MATLAB software type `int32`.

`value = winqueryreg('rootkey', 'subkey')` returns a value in `rootkey\subkey` that has no value name property.

**Note**
The literal `name` argument and the `rootkey` argument are case-sensitive. The `subkey` and `valname` arguments are not.

**Remarks**

This function works only for the following registry value types:

- strings (REG_SZ)
- expanded strings (REG_EXPAND_SZ)
- 32-bit integer (REG_DWORD)

**Examples**

**Example 1**

Get the value of `CLSID` for the MATLAB sample Microsoft COM control `mwsampctrl.2`:

```matlab
winqueryreg 'HKEY_CLASSES_ROOT' 'mwsamp.mwsampctrl.2\clsid'
```
Example 2

Get a list in variable `mousechar` for registry subkey `Mouse`, which is under subkey `Control Panel`, which is under root key `HKEY_CURRENT_USER`.

```matlab
mousechar = winqueryreg('name', 'HKEY_CURRENT_USER', ...
    'control panel\mouse');
```

For each name in the `mousechar` list, get its value from the registry and then display the name and its value:

```matlab
for k=1:length(mousechar)
    setting = winqueryreg('HKEY_CURRENT_USER', ...
        'control panel\mouse', mousechar{k});
    str = sprintf('%s = %s', mousechar{k}, num2str(setting));
    disp(str)
end
```

ActiveWindowTracking = 0
DoubleClickHeight = 4
DoubleClickSpeed = 830
DoubleClickWidth = 4
MouseSpeed = 1
MouseThreshold1 = 6
MouseThreshold2 = 10
SnapToDefaultButton = 0
SwapMouseButtons = 0
**Purpose**  
Determine whether file contains 1-2-3 WK1 worksheet

**Syntax**  
`[extens, typ] = wk1finfo(filename)`

**Description**  
`[extens, typ] = wk1finfo(filename)` returns the string 'WK1' in `extens`, and '1-2-3 Spreadsheet' in `typ` if the file `filename` contains a readable worksheet. The `filename` input is a string enclosed in single quotes.

**Examples**  
This example returns information on spreadsheet file `matA.wk1`:

```matlab
[extens, typ] = wk1finfo('matA.wk1')
```

```
extens =
    WK1
typ =
    123 Spreadsheet
```

**See Also**  
wk1read, wk1write, csvread, csvwrite
**Purpose**
Read Lotus 1-2-3 WK1 spreadsheet file into matrix

**Syntax**

\[
M = wk1read(filename)
M = wk1read(filename,r,c)
M = wk1read(filename,r,c,range)
\]

**Description**

\(M = wk1read(filename)\) reads a Lotus1-2-3 WK1 spreadsheet file into the matrix \(M\). The filename input is a string enclosed in single quotes.

\(M = wk1read(filename,r,c)\) starts reading at the row-column cell offset specified by \((r,c)\). \(r\) and \(c\) are zero based so that \(r=0\), \(c=0\) specifies the first value in the file.

\(M = wk1read(filename,r,c,range)\) reads the range of values specified by the parameter range, where range can be

- A four-element vector specifying the cell range in the format

  \[
  \begin{bmatrix}
  \text{upper_left_row} & \text{upper_left_col} & \text{lower_right_row} & \text{lower_right_col}
  \end{bmatrix}
  \]

- A cell range specified as a string, for example, 'A1...C5'
- A named range specified as a string, for example, 'Sales'

**Examples**
Create a 8-by-8 matrix \(A\) and export it to Lotus spreadsheet matA.wk1:

\[
A = \begin{bmatrix}
\end{bmatrix}
\]

\[
A =
\]
To read in a limited block of the spreadsheet data, specify the upper left row and column of the block using zero-based indexing:

```matlab
M = wk1read('matA.wk1', 3, 2)
```

```
M =
33  34  35  36  37  38
43  44  45  46  47  48
53  54  55  56  57  58
63  64  65  66  67  68
73  74  75  76  77  78
```

To select a more restricted block of data, you can specify both the upper left and lower right corners of the block you want imported. Read in a range of values from row 4, column 3 (defining the upper left corner) to row 6, column 6 (defining the lower right corner). Note that, unlike the second and third arguments, the range argument [4 3 6 6] is one-based:

```matlab
M = wk1read('matA.wk1', 3, 2, [4 3 6 6])
```

```
M =
33  34  35  36
43  44  45  46
53  54  55  56
```

**See Also**

`wk1write`
**Purpose**

Write matrix to Lotus 1-2-3 WK1 spreadsheet file

**Syntax**

```
wk1write(filename,M)
wk1write(filename,M,r,c)
```

**Description**

`wk1write(filename,M)` writes the matrix `M` into a Lotus1-2-3 WK1 spreadsheet file named `filename`. The `filename` input is a string enclosed in single quotes.

`wk1write(filename,M,r,c)` writes the matrix starting at the spreadsheet location `(r,c)`. `r` and `c` are zero based so that `r=0, c=0` specifies the first cell in the spreadsheet.

**Examples**

Write a 4-by-5 matrix `A` to spreadsheet file `matA.wk1`. Place the matrix with its upper left corner at row 2, column 3 using zero-based indexing:

```
A =
    1     2     3     4     5
   11    12    13    14    15
   21    22    23    24    25
   31    32    33    34    35

wk1write('matA.wk1', A, 2, 3)
```

```
M = wk1read('matA.wk1')
M =
```
See Also: wk1read, dlmwrite, dlmread, csvwrite, csvread
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Open Workspace browser to manage workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GUI</strong></td>
<td>As an alternative to the <code>workspace</code> function, select <code>Desktop &gt; Workspace</code> in the MATLAB desktop.</td>
</tr>
<tr>
<td><strong>Alternatives</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td><code>workspace</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The <code>workspace</code> function displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the workspace in MATLAB. It provides a graphical representation of the <code>whos</code> display, and allows you to perform the equivalent of the <code>clear</code>, <code>load</code>, <code>open</code>, and <code>save</code> functions. The Workspace browser also displays and automatically updates statistical calculations for each variable, which you can choose to show or hide.</td>
</tr>
</tbody>
</table>
You can edit a value directly in the Workspace browser for small numeric and character arrays. To see and edit a graphical representation of larger variables and for other classes, double-click the variable in the Workspace browser. The variable displays in the Variable Editor, where you can view the full contents and make changes.
See Also

openvar, who

“MATLAB Workspace”
xlabel, ylabel, zlabel

**Purpose**

Label x-, y-, and z-axis

**GUI Alternative**

To control the presence and appearance of axis labels on a graph, use the Property Editor, one of the plotting tools 🔄. For details, see The Property Editor in the MATLAB Graphics documentation.

**Syntax**

```
xlabel('string')
xlabel(fname)
xlabel(...,'PropertyName',PropertyValue,...)
xlabel(axes_handle,...)
h = xlabel(...)
```

```
ylabel(...)  
ylabel(axes_handle,...)  
h = ylabel(...)
```

```
zlabel(...)  
zlabel(axes_handle,...)  
h = zlabel(...)
```

**Description**

Each axes graphics object can have one label for the x-, y-, and z-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

- `xlabel('string')` labels the x-axis of the current axes.
- `xlabel(fname)` evaluates the function `fname`, which must return a string, then displays the string beside the x-axis.
- `xlabel(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the text graphics object created by `xlabel`. 
xlabel(axes_handle,...), ylabel(axes_handle,...), and zlabel(axes_handle,...) plot into the axes with handle axes_handle instead of the current axes (gca).

h = xlabel(...), h = ylabel(...), and h = zlabel(...) return the handle to the text object used as the label.

ylabel(...) and zlabel(...) label the y-axis and z-axis, respectively, of the current axes.

Remarks

Reissuing an xlabel, ylabel, or zlabel command causes the new label to replace the old label.

For three-dimensional graphics, MATLAB puts the label in the front or side, so that it is never hidden by the plot.

Examples

Create a multiline label for the x-axis using a multiline cell array:

```
xlabel({'first line';'second line'})
```

Create a bold label for the y-axis that contains a single quote:

```
ylabel('George''s Popularity','fontsize',12,'fontweight','b')
```

See Also

strings, text, title

“Annotating Plots” on page 1-94 for related functions

“Adding Axis Labels to Graphs” for more information about labeling axes
**xlim, ylim, zlim**

**Purpose**
Set or query axis limits

**GUI Alternative**
To control the upper and lower axis limits on a graph, use the Property Editor, one of the plotting tools. For details, see The Property Editor in the MATLAB Graphics documentation.

**Syntax**

```
xlim
xlimit([xmin xmax])
xlim('mode')
xlim('auto')
xlim('manual')
xlim(axes_handle,...)
```

Note that the syntax for each of these three functions is the same; only the `xlim` function is used for simplicity. Each operates on the respective x-, y-, or z-axis.

**Description**

- `xlim` with no arguments returns the respective limits of the current axes.
- `xlim([xmin xmax])` sets the axis limits in the current axes to the specified values.
- `xlim('mode')` returns the current value of the axis limits mode, which can be either `auto` (the default) or `manual`.
- `xlim('auto')` sets the axis limit mode to `auto`.
- `xlim('manual')` sets the respective axis limit mode to `manual`.
- `xlim(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, these functions operate on the current axes.

**Remarks**

`xlim`, `ylim`, and `zlim` set or query values of the axes object `XLim`, `YLim`, `ZLim`, and `XLimMode`, `YLimMode`, `ZLimMode` properties.

When the axis limit modes are `auto` (the default), MATLAB uses limits that span the range of the data being displayed and are round numbers.
Setting a value for any of the limits also sets the corresponding mode to `manual`. Note that high-level plotting functions like `plot` and `surf` reset both the modes and the limits. If you set the limits on an existing graph and want to maintain these limits while adding more graphs, use the `hold` command.

**Examples**

This example illustrates how to set the $x$- and $y$-axis limits to match the actual range of the data, rather than the rounded values of $[-2\ 3]$ for the $x$-axis and $[-2\ 4]$ for the $y$-axis originally selected by MATLAB.

```matlab
[x,y] = meshgrid([-1.75:.2:3.25]);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z)
xlim([-1.75 3.25])
ylim([-1.75 3.25])
```

![Example graph](image)
See Also

axis

The axes properties XLim, YLim, ZLim

“Aspect Ratio and Axis Limits” on page 1-107 for related functions

Understanding Axes Aspect Ratio for more information on how axis limits affect the axes
**Purpose**
Determine whether file contains a Microsoft Excel spreadsheet

**Syntax**

```matlab
typ = xlsfinfo(filename)
[typ, desc] = xlsfinfo(filename)
[typ, desc, fmt] = xlsfinfo(filename)
xlsfinfo filename
```

**Description**

`typ = xlsfinfo(filename)` returns the string 'Microsoft Excel Spreadsheet' if the file specified by `filename` is an Excel file that can be read by the MATLAB `xlsread` function. Otherwise, `typ` is the empty string, (''). The `filename` input is a string enclosed in single quotation marks.

`[typ, desc] = xlsfinfo(filename)` returns in `desc` a cell array of strings containing the names of each spreadsheet in the file. If a spreadsheet is unreadable, the cell in `desc` that represents that spreadsheet contains an error message.

`[typ, desc, fmt] = xlsfinfo(filename)` returns in the `fmt` output a string containing the Excel-reported file format. On UNIX systems, or on Windows systems without Excel software installed, `xlsfinfo` returns `fmt` as an empty string, ('').

`xlsfinfo filename` is the command format for `xlsfinfo`. It returns only the first output, `typ`, assigning it to the MATLAB default variable `ans`.

**Remarks**
If your system has Excel for Windows installed, `xlsfinfo` uses the COM server to obtain information. This server is part of the typical installation of Excel for Windows. If the COM server is unavailable, `xlsfinfo` returns a warning indicating that it cannot start an ActiveX server. To establish connectivity with the COM server, you might need to reinstall your Excel software.

**Examples**
Get information about an .xls file:

```matlab
[typ, desc, fmt] = xlsfinfo('myaccount.xls')
```
typ =
    Microsoft Excel Spreadsheet

desc =
    'Sheet1'    'Income'    'Expenses'

fmt =
    xlWorkbookNormal

Export the .xls file to comma-separated value (CSV) format. Use xlsinfo to see the format of the exported file:

    [typ, desc, fmt] = xlsinfo('myaccount.csv');
    fmt

    fmt =
        xlCSV

Export the .xls file to HTML format. xlsinfo returns the following format string:

    [typ, desc, fmt] = xlsinfo('myaccount.html');
    fmt

    fmt =
        xlHtml

Export the .xls file to XML format. xlsinfo returns the following format string:

    [typ, desc, fmt] = xlsinfo('myaccount.xml');
    fmt

    fmt =
        x1XMLSpreadsheet

See Also
    xlsread, xlswrite
Purpose
Read Microsoft Excel spreadsheet file

Syntax
num = xlsread(filename)
num = xlsread(filename, -1)
num = xlsread(filename, sheet)
num = xlsread(filename, range)
num = xlsread(filename, sheet, range)
num = xlsread(filename, sheet, range, 'basic')
num = xlsread(filename, ..., functionhandle)
[num, txt] = xlsread(filename, ...)
[num, txt, raw] = xlsread(filename, ...)
[num, txt, raw, X] = xlsread(filename, ..., functionhandle)
xlsread filename sheet range basic

Description
num = xlsread(filename) returns numeric data in double array
num from the first sheet in the Microsoft Excel spreadsheet file named
filename. The filename argument is a string enclosed in single
quotation marks.

xlsread ignores any outer rows or columns of the spreadsheet that
contain no numeric data. If there are single or multiple nonnumeric
rows at the top or bottom, or single or multiple nonnumeric columns to
the left or right, xlsread does not include these rows or columns in
the output. For example, xlsread ignores one or more header lines
appearing at the top of a spreadsheet.

Any inner rows or columns in which some or all cells contain
nonnumeric data are not ignored. Instead, xlsread assigns a value of
NaN to the nonnumeric cells.

num = xlsread(filename, -1) opens the file filename in an Excel
window, enabling you to interactively select the worksheet to read and
the range of data on that worksheet to import.

To import an entire worksheet, first select the sheet in the Excel
window, and then click the OK button in the Data Selection Dialog box.
To import a certain range of data from the sheet, select the worksheet in
the Excel window, drag and drop the mouse over the desired range, and
then click OK. (See “COM Server Requirements” on page 2-4210 below.)
num = xlsread(filename, sheet) reads the specified worksheet, where sheet is either a positive, double scalar value or a quoted string containing the sheet name. To determine the names of the sheets in a spreadsheet file, use xlsinfo.

num = xlsread(filename, range) reads data from a specific rectangular region of the default worksheet (Sheet1). (See “COM Server Requirements” on page 2-4210 below.)

Specify range using the syntax 'C1:C2', where C1 and C2 are two opposing corners that define the region to be read. For example, 'D2:H4' represents the 3-by-5 rectangular region between the two corners D2 and H4 on the worksheet. The range input is not case sensitive and uses Excel A1 reference style. For more information on this reference style, see Excel help.

Note If you specify only two inputs, xlsread must decide whether the second input refers to a sheet or a range. To specify a range (even a range of a single cell), include a colon character in the input string (e.g., 'D2:H4'). If you do not include a colon character (e.g., 'sales' or 'D2'), xlsread interprets the second input as the name or index of a worksheet.

num = xlsread(filename, sheet, range) reads data from a specific rectangular region (range) of the worksheet specified by sheet. If you specify both sheet and range, range can refer to a named range that you defined in the Excel file. (For more information on named ranges, see the Excel help.) See the previous two syntax formats for further explanation of the sheet and range inputs. (Also, see “COM Server Requirements” on page 2-4210 below.)

num = xlsread(filename, sheet, range, 'basic') imports data from the spreadsheet in basic import mode. xlsread uses this mode on systems where Excel software is not installed. Import ability is limited. xlsread ignores the value for range and, consequently, imports the whole active range of a sheet. (You can set range to the empty string
num = xlsread(filename, ..., functionhandle) calls the function associated with functionhandle just prior to obtaining spreadsheet values. This enables you to operate on the spreadsheet data (for example, convert it to a numeric type) before reading it in. (See “COM Server Requirements” on page 2-4210 below.)

You can write your own custom function and pass a handle to this function to xlsread. When xlsread executes, it reads from the spreadsheet, executes your function on the data read from the spreadsheet, and returns the final results to you. When xlsread calls your function, it passes a range interface from the Excel application to provide access to the data read from the spreadsheet. Your function must include this interface both as an input and output argument. Example 5 below shows how you might use this syntax.

For more information, see “Function Handles” in the MATLAB Programming Fundamentals documentation.

[num, txt]= xlsread(filename, ...) returns numeric data in array num and text data in cell array txt. All cells in txt that correspond to numeric data contain the empty string.

[num, txt, raw] = xlsread(filename, ...) returns numeric and text data in num and txt, and unprocessed cell content in cell array raw, which contains both numeric and text data. (See “COM Server Requirements” on page 2-4210 below.)

If the Excel file includes cells with undefined values (such as '#N/A'), xlsread returns these values as '#N/A' in the txt output, and as 'ActiveX VT_ERROR:' in the raw output.

[num, txt, raw, X] = xlsread(filename, ..., functionhandle) calls the function associated with functionhandle just prior to reading from the spreadsheet file. This syntax returns one additional output X from the function mapped to by functionhandle. Example 6 below shows how you might use this syntax. (See “COM Server Requirements” on page 2-4210 below.)
xlsread filename sheet range basic is an example of the command format for xlsread, showing its usage with all input arguments specified. When using this format, you must specify sheet as a string, (for example, Income or Sheet4) and not a numeric index. If the sheet name contains space characters, then quotation marks are required around the string, (for example, 'Income 2002').

**Remarks**

**COM Server Requirements**

The typical installation of Excel for Windows includes the ability to start a COM server. With Excel for Windows installed, you can use xlsread to read any file format recognized by your version of Excel, including XLS, XLSX, XLSB, XLSM, and HTML-based formats.

If your system does not have Excel for Windows installed, or MATLAB cannot access the COM server, xlsread operates in basic mode. In this mode, xlsread only reads XLS files.

The following five syntax formats are supported only on computer systems able to start a COM server from a MATLAB session. They are not supported in basic mode.

```matlab
num = xlsread(filename, -1)
num = xlsread(filename, 'range')
num = xlsread(filename, sheet, 'range')
num = xlsread(filename, ..., functionhandle)
[num, txt, raw, opt] = xlsread(filename, ..., functionhandle)
```

**Handling Excel Date Values**

MATLAB functions import all formatted dates as strings. To import a numeric date, the date field in Excel must have a numeric format.

Both Excel and MATLAB applications represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel and MATLAB use different reference dates:

<table>
<thead>
<tr>
<th>Application</th>
<th>Reference Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>January 0, 0000</td>
</tr>
<tr>
<td>Application</td>
<td>Reference Date</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Excel for Windows</td>
<td>January 1, 1900</td>
</tr>
<tr>
<td>Excel for the Macintosh</td>
<td>January 2, 1904</td>
</tr>
</tbody>
</table>

Therefore, you must convert any numeric date that you import before you process it in MATLAB. For more information, see “Converting Dates” in the MATLAB Programming Fundamentals documentation.

Consider using the function handle parameter for this conversion, discussed in the Syntax Description and in Example 5 and Example 6.

**Examples**

**Example 1 — Reading Numeric Data**

The Microsoft Excel spreadsheet file `testdata1.xls` contains this data:

```
1   6
2   7
3   8
4   9
5  10
```

To read this data into MATLAB, use this command:

```
A = xlsread('testdata1.xls')
```

```
A =
   1   6
   2   7
   3   8
   4   9
   5  10
```

**Example 2 — Handling Text Data**

The Microsoft Excel spreadsheet file `testdata2.xls` contains a mix of numeric and text data:

```
1   6
2   7
3   8
```
xlsread

4  9
5  text

xlsread puts a NaN in place of the text data in the result:

A = xlsread('testdata2.xls')
A =

1   6
2   7
3   8
4   9
5  NaN

Example 3 — Selecting a Range of Data

To import only rows 4 and 5 from worksheet 1, specify the range as 'A4:B5':

A = xlsread('testdata2.xls', 1, 'A4:B5')

A =

4   9
5  NaN

Example 4 — Handling Files with Row or Column Headers

A Microsoft Excel spreadsheet labeled Temperatures in file tempdata.xls contains two columns of numeric data with text headers for each column:

<table>
<thead>
<tr>
<th>Time</th>
<th>Temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>98</td>
</tr>
<tr>
<td>13</td>
<td>99</td>
</tr>
<tr>
<td>14</td>
<td>97</td>
</tr>
</tbody>
</table>

If you want to import only the numeric data, use xlsread with a single return argument. Specify the filename and sheet name as inputs.

ndata = xlsread('tempdata.xls', 'Temperatures')
To import both the numeric data and the text data, specify two return values for `xlsread`:

```matlab
[ndata, headertext] = xlsread('tempdata.xls', 'Temperatures')
```

```plaintext
ndata =
  12  98
  13  99
  14  97
```

```plaintext
headertext =
  'Time'    'Temp'
```

**Example 5 — Passing a Function Handle**

This example calls `xlsread` twice, the first time as a simple read from a file, and the second time requesting that `xlsread` execute some user-defined modifications on the data prior to returning the results of the read. A user-written function, `setMinMax`, that you pass as a function handle in the call to `xlsread`, performs these modifications. When `xlsread` executes, it reads from the spreadsheet, executes the function on the data read from the spreadsheet, and returns the final results to you.

**Note** The function passed to `xlsread` operates on the copy of the data read from the spreadsheet. It does not modify data in the spreadsheet itself.
Read a 10-by-3 numeric array from Excel spreadsheet `testsheet.xls` with a simple `xlsread` statement that does not pass a function handle. The returned values range from -587 to +4,149:

```matlab
arr = xlsread('testsheet.xls')
arr =
    1.0e+003 *
   1.0020    4.1490    0.2300
   1.0750    0.1220   -0.4550
  -0.0301    3.0560    0.2471
   0.4070    0.1420   -0.2472
  2.1160   -0.0557   -0.5870
   0.4040    2.9280    0.0265
  0.1723    3.4440    0.1112
  4.1180    0.1820    2.8630
  0.9000    0.0573    1.9750
  0.0163    0.2000   -0.0223
```

In preparation for the second part of this example, write a function `setMinMax` that restricts the values returned from the read to be in the range of 0 to 2000. You need to pass this function in the call to `xlsread`, which then executes the function on the data it has read before returning it to you.

When `xlsread` calls your function, it passes an Excel range interface to provide access to the data read from the spreadsheet. This is shown as `DataRange` in this example. Your function must include this interface both as an input and output argument. The output argument allows your function to pass modified data back to `xlsread`:

```matlab
function [DataRange] = setMinMax(DataRange)
maxval = 2000;  minval = 0;

for k = 1:DataRange.Count
    v = DataRange.Value{k};
    if v > maxval || v < minval
        if v > maxval
            DataRange.Value{k} = maxval;
```
else
    DataRange.Value{k} = minval;
end
end
end

Now call `xlsread`, passing a function handle for the `setMinMax` function as the final argument. After this call, all values are between 0 and 2000:

```matlab
arr = xlsread('testsheet.xls', '', '', '', @setMinMax)
```
```
arr =
1.0e+003 *
 1.0020 2.0000 0.2300
 1.0750 0.1220 0
  0 2.0000 0.2471
 0.4070 0.1420 0
 2.0000 0 0
 0.4040 2.0000 0.0265
 0.1723 2.0000 0.1112
 2.0000 0.1820 2.0000
 0.9000 0.0573 1.9750
 0.0163 0.2000 0
```

**Example 6 — Passing a Function Handle with Additional Output**

This example adds onto the previous one by returning an additional output from the call to `setMinMax`. Modify the function so that it not only limits the range of values returned, but also returns the indices of the altered elements. Return this information in a new output argument, `indices`:

```matlab
function [DataRange, indices] = setMinMax(DataRange)
maxval = 2000; minval = 0;
indices = [];

for k = 1:DataRange.Count
    v = DataRange.Value{k};
    if v > maxval || v < minval
```
if \( v > \text{maxval} \)
    DataRange.Value\{k\} = \text{maxval};
else
    DataRange.Value\{k\} = \text{minval};
end
indices = [indices k];
end
end

When you call xlsread this time, account for the three initial outputs, and add a fourth called idx to accept the indices returned from setMinMax:

\[
\text{[arr txt raw idx]} = \text{xlsread('testsheet.xls', ...}
\]
\[
\quad \quad \quad \quad \quad ', '', '', \text{, @setMinMax)};
\]

idx
\[
\text{idx} =
\]
\[
\begin{align*}
3 & \quad 5 & \quad 8 & \quad 11 & \quad 13 & \quad 15 & \quad 16 & \quad 17 & \quad 22 & \quad 24 & \quad 25 & \quad 28 & \quad 30 \\
\end{align*}
\]
arr
\[
\text{arr} =
\]
\[
\begin{align*}
1.0e+003 \cdot & \\
1.0020 & \quad 2.0000 & \quad 0.2300 \\
1.0750 & \quad 0.1220 & \quad 0 & \\
0 & \quad 2.0000 & \quad 0.2471 \\
0.4070 & \quad 0.1420 & \quad 0 & \\
2.0000 & \quad 0 & \quad 0 \\
0.4040 & \quad 2.0000 & \quad 0.0265 \\
0.1723 & \quad 2.0000 & \quad 0.1112 \\
2.0000 & \quad 0.1820 & \quad 2.0000 \\
0.9000 & \quad 0.0573 & \quad 1.9750 \\
0.0163 & \quad 0.2000 & \quad 0 \\
\end{align*}
\]

See Also
xlswrite, xlsfinfo, importdata, uimport, textscan,
function_handle
**Purpose**

Write Microsoft Excel spreadsheet file

**Syntax**

\[ \text{xlswrite(filename, M)} \]
\[ \text{xlswrite(filename, M, sheet)} \]
\[ \text{xlswrite(filename, M, range)} \]
\[ \text{xlswrite(filename, M, sheet, range)} \]
\[ \text{status} = \text{xlswrite(filename, ...)} \]
\[ \text{[status, message]} = \text{xlswrite(filename, ...)} \]
\[ \text{xlswrite filename M sheet range} \]

**Description**

`xlswrite(filename, M)` writes matrix `M` to the Excel file `filename`. The `filename` input is a string enclosed in single quotation marks, and should include the file extension. The matrix `M` is an \( m \)-by-\( n \) numeric or character array. `xlswrite` writes the matrix data to the first worksheet in the file, starting at cell A1.

The matrix `M` can also be an \( m \)-by-\( n \) cell array if each cell includes a single element (see Example 2). To write cell arrays containing more than one element in each cell, use low-level export functions.

If `filename` does not exist, `xlswrite` creates a new file. The file extension you provide as part of `filename` determines the Excel format that `xlswrite` uses for the new file. An extension of `.xls` creates a worksheet compatible with Excel 97-2003 software. Use extensions `.xlsx`, `.xlsb`, or `.xlsm` to create worksheets in Excel 2007 file formats. The maximum size of the matrix `M` depends on the associated Excel version. (For more information on Excel specifications and limits, see Excel help.)

`xlswrite(filename, M, sheet)` writes matrix `M` to the specified worksheet `sheet` in the file `filename`. The `sheet` argument can be either a positive, double scalar value representing the worksheet index, or a quoted string containing the sheet name. The `sheet` argument cannot contain a colon.

If `sheet` does not exist, `xlswrite` adds a new sheet at the end of the worksheet collection. If `sheet` is an index larger than the number of worksheets, `xlswrite` appends empty sheets until the number of
worksheets in the workbook equals sheet. In either case, xlswrite generates a warning indicating that it has added a new worksheet.

xlswrite(filename, M, range) writes matrix M to a rectangular region specified by range in the first worksheet of the file filename.

Specify range using the syntax 'C1:C2', where C1 and C2 are two opposing corners that define the region to write. For example, the range 'D2:H4' represents the 3-by-5 rectangular region between the two corners D2 and H4 on the worksheet. The range input is not case sensitive and uses the Excel A1 reference style. (For more information on this reference style, see Excel help.) xlswrite does not recognize named ranges.

The size defined by range should fit the size of M. If range is larger than the size of M, Excel software fills the remainder of the region with #N/A. If range is smaller than the size of M, xlswrite writes only the submatrix that fits into range to the file specified by filename.

**Note** If you specify only three inputs, xlswrite must decide whether the third input refers to a sheet or a range. To specify a range, include a colon character in the input string (such as 'D2:H4'). If you do not include a colon character (such as 'sales' or 'D2'), xlswrite interprets the third input as a value for sheet.

xlswrite(filename, M, sheet, range) writes matrix M to a rectangular region specified by range in worksheet sheet of the file filename. If you specify both sheet and range, the range can either fit the size of M or contain only the first cell (such as 'A2'). See the previous two syntax formats for further explanation of the sheet and range inputs.

status = xlswrite(filename, ...) returns the completion status of the write operation in status. If the write completes successfully, status is equal to logical 1 (true). Otherwise, status is logical 0 (false). Unless you specify an output parameter, xlswrite does not display a status value in the Command Window.
[status, message] = xlswrite(filename, ...) returns any warning or error message generated by the write operation in the MATLAB structure message. The message structure has two fields:

- message — String containing the text of the warning or error message
- identifier — String containing the message identifier for the warning or error

xlswrite filename M sheet range is the command format for xlswrite, showing its usage with all input arguments specified. When using this format, you must specify sheet as a string (for example, Income or Sheet4). If the sheet name contains space characters, then you must place quotation marks around the string (for example, 'Income 2002').

Remarks

Full functionality of xlswrite depends on the use of the Microsoft Excel COM server. The typical installation of Excel for Windows includes access to this server. If your system does not have Excel for Windows installed, or if the COM server is unavailable, xlswrite:

- Writes matrix M as a text file in comma-separated value (CSV) format.
- Ignores the sheet and range arguments.
- Generates an error if the input matrix M is a cell array.

If your system has Microsoft Office 2003 software installed, but you want to create a file in an Excel 2007 format, you must install the Office 2007 Compatibility Pack.

Both Excel and MATLAB applications represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel and MATLAB use different reference dates:

<table>
<thead>
<tr>
<th>Application</th>
<th>Reference Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>January 0, 0000</td>
</tr>
</tbody>
</table>
Application Reference Date
Excel for Windows January 1, 1900
Excel for the Macintosh January 2, 1904

For more information, see “Converting Dates” in the MATLAB Programming Fundamentals documentation.

Examples

Example 1 — Writing Numeric Data to the Default Worksheet
Write a 7-element vector to Microsoft Excel file testdata.xls. By default, xlswrite writes the data to cells A1 through G1 in the first worksheet in the file:

    xlswrite('testdata.xls', [12.7 5.02 -98 63.9 0 -.2 56])

Example 2 — Writing Mixed Data to a Specific Worksheet
This example writes the following mixed text and numeric data to the file tempdata.xls:

    d = {'Time', 'Temp'; 12 98; 13 99; 14 97};

Call xlswrite, specifying the worksheet labeled Temperatures, and the region within the worksheet to write the data to. xlswrite writes the 4-by-2 matrix to the rectangular region that starts at cell E1 in its upper left corner:

    s = xlswrite('tempdata.xls', d, 'Temperatures', 'E1')
    s =
        1

The output status s shows that the write operation succeeded. The data appears as shown here in the output file:

    Time   Temp
    12     98
    13     99
    14     97
Example 3 — Appending a New Worksheet to the File

Now write the same data to a worksheet that doesn’t yet exist in tempdata.xls. In this case, xlswrite appends a new sheet to the workbook, calling it by the name you supplied in the sheets input argument, 'NewTemp'. xlswrite displays a warning indicating that it has added a new worksheet to the file:

```matlab
xlswrite('tempdata.xls', d, 'NewTemp', 'E1')
Warning: Added specified worksheet.
```

If you don’t want to see these warnings, you can turn them off with this command:

```matlab
warning off MATLAB:xlswrite:AddSheet
```

Now try the write command again, this time creating another new worksheet, NewTemp2. Although the message does not appear this time, you can still retrieve it and its identifier from the second output argument, msg:

```matlab
[stat msg] = xlswrite('tempdata.xls', d, 'NewTemp2', 'E1');
```

```matlab
msg
msg =
message: 'Added specified worksheet.'
identifier: 'MATLAB:xlswrite:AddSheet'
```

See Also

xlswrite, xlsfinfo
xmlread

**Purpose**
Parse XML document and return Document Object Model node

**Syntax**
DOMnode = xmlread(filename)

**Description**
DOMnode = xmlread(filename) reads a URL or filename and returns a Document Object Model node representing the parsed document. The filename input is a string enclosed in single quotes. The node can be manipulated by using standard DOM functions.

A properly parsed document displays to the screen as

```matlab
xDoc = xmlread(...)  
xDoc =  
    [#document: null]
```

**Remarks**

**Examples**

**Example 1**
All XML files have a single root element. Some XML files declare a preferred schema file as an attribute of this element. Use the `getAttribute` method of the DOM node to get the name of the preferred schema file:

```matlab
xDoc = xmlread(fullfile(matlabroot, ...  
    'toolbox/matlab/general/info.xml'));

xRoot = xDoc.getDocumentElement;
schemaURL = ...
    char(xRoot.getAttribute('xsi:noNamespaceSchemaLocation'))

    schemaURL =  
        http://www.mathworks.com/namespace/info/v1/info.xsd
```
Example 2

Each info.xml file on the MATLAB path contains several listitem elements with a label and callback element. This script finds the callback that corresponds to the label 'Plot Tools':

```matlab
infoLabel = 'Plot Tools';
infoCbk = '';
itemFound = false;

xDoc = xmlread(fullfile(matlabroot, ...
    'toolbox/matlab/general/info.xml'));

% Find a deep list of all listitem elements.
allListItems = xDoc.getElementsByTagName('listitem');

% Note that the item list index is zero-based.
for k = 0:allListItems.getLength-1
    thisListItem = allListItems.item(k);
    childNode = thisListItem.getFirstChild;
    while ~isempty(childNode)
        % Filter out text, comments, and processing instructions.
        if childNode.getNodeType == childNode.ELEMENT_NODE
            % Assume that each element has a single
            % org.w3c.dom.Text child.
            childText = char(childNode.getFirstChild.getData);

            switch char(childNode.getTagName)
                case 'label';
                    itemFound = strcmp(childText, infoLabel);
                case 'callback';
                    infoCbk = childText;
            end
        end % End IF
        childNode = childNode.getNextSibling;
    end % End WHILE
end % End FOR
```
if itemFound
    break;
else
    infoCbk = ''; 
end
end % End FOR

disp(sprintf('Item "%s" has a callback of "%s", ... 
            infoLabel, infoCbk))

Example 3

This function parses an XML file using methods of the DOM node returned by xmlread, and stores the data it reads in the Name, Attributes, Data, and Children fields of a MATLAB structure:

function theStruct = parseXML(filename)
% PARSEXML Convert XML file to a MATLAB structure.
try
    tree = xmlread(filename);
catch
    error('Failed to read XML file %s.',filename);
end

% Recurse over child nodes. This could run into problems
% with very deeply nested trees.
try
    theStruct = parseChildNodes(tree);
catch
    error('Unable to parse XML file %s.',filename);
end

% ----- Subfunction PARSECILDNODES ----- 
function children = parseChildNodes(theNode)
% Recurse over node children.
children = [];
if theNode.hasChildNodes
childNodes = theNode.getChildNodes;
numChildNodes = childNodes.getLength;
allocCell = cell(1, numChildNodes);

children = struct( ...
    'Name', allocCell, 'Attributes', allocCell, ... 
    'Data', allocCell, 'Children', allocCell);

for count = 1:numChildNodes
    theChild = childNodes.item(count-1);
    children(count) = makeStructFromNode(theChild);
end
end

% ----- Subfunction MAKESTRUCTFROMNODE ----- 
function nodeStruct = makeStructFromNode(theNode)
% Create structure of node info.

nodeStruct = struct( ... 
    'Name', char(theNode.getNodeName), ... 
    'Attributes', parseAttributes(theNode), ... 
    'Data', '', ... 
    'Children', parseChildNodes(theNode));

if any(strcmp(methods(theNode), 'getData'))
    nodeStruct.Data = char(theNode.getData);
else
    nodeStruct.Data = ''; 
end

% ----- Subfunction PARSEATTRIBUTES ----- 
function attributes = parseAttributes(theNode)
% Create attributes structure.

attributes = [];
if theNode.hasAttributes
    theAttributes = theNode.getAttributes;
    for each attribute
        attributes = [attributes, ... ];
    end
end

numAttributes = theAttributes.getLength;
allocCell = cell(1, numAttributes);
attributes = struct('Name', allocCell, 'Value', ...
    allocCell);

for count = 1:numAttributes
    attrib = theAttributes.item(count-1);
    attributes(count).Name = char(attrib.getName);
    attributes(count).Value = char(attrib.getValue);
end

See Also  xmlwrite, xslt
Purpose
Serialize XML Document Object Model node

Syntax
xmlwrite(filename, DOMnode)
str = xmlwrite(DOMnode)

Description
xmlwrite(filename, DOMnode) serializes the Document Object Model node DOMnode to the file specified by filename. The filename input is a string enclosed in single quotes.

str = xmlwrite(DOMnode) serializes the Document Object Model node DOMnode and returns the node tree as a string, s.

Remarks

Example
% Create a sample XML document.
docNode = com.mathworks.xml.XMLUtils.createDocument...
   ('root_element')
docRootNode = docNode.getDocumentElement;
for i=1:20
    thisElement = docNode.createElement('child_node');
    thisElement.appendChild...
        (docNode.createTextNode(sprintf('%i',i)));
    docRootNode.appendChild(thisElement);
end
docNode.appendChild(docNode.createComment('this is a comment'));

% Save the sample XML document.
xmlFileName = [tempname,'.xml'];
xmlwrite(xmlFileName,docNode);
edit(xmlFileName);

See Also
xmlread, xslt
**Purpose**
Logical exclusive-OR

**Syntax**
\[ C = \text{xor}(A, B) \]

**Description**
\( C = \text{xor}(A, B) \) performs an exclusive OR operation on the corresponding elements of arrays \( A \) and \( B \). The resulting element \( C(i,j,...) \) is logical true (1) if \( A(i,j,...) \) or \( B(i,j,...) \), but not both, is nonzero.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>Zero</td>
<td>Nonzero</td>
<td>1</td>
</tr>
<tr>
<td>Nonzero</td>
<td>Zero</td>
<td>1</td>
</tr>
<tr>
<td>Nonzero</td>
<td>Nonzero</td>
<td>0</td>
</tr>
</tbody>
</table>

**Examples**
Given \( A = [0 \ 0 \ \text{pi} \ \text{eps}] \) and \( B = [0 \ -2.4 \ 0 \ 1] \), then

\[
C = \text{xor}(A,B)
C = \\
\begin{bmatrix}
0 & 1 & 1 & 0
\end{bmatrix}
\]

To see where either \( A \) or \( B \) has a nonzero element and the other matrix does not,

\[ \text{spy} (\text{xor}(A,B)) \]

**See Also**
all, any, find, Elementwise Logical Operators, Short-Circuit Logical Operators
**Purpose**
Transform XML document using XSLT engine

**Syntax**

\[
\text{result} = \text{xslt}(\text{source, style, dest})
\]
\[
[\text{result,style}] = \text{xslt}(\ldots)
\]
\[
\text{xslt}(\ldots,'-\text{web}')
\]

**Description**

\text{result} = \text{xslt}(\text{source, style, dest}) transforms an XML document using a stylesheet and returns the resulting document's URL. The function uses these inputs, the first of which is required:

- \text{source} is the filename or URL of the source XML file. \text{source} can also specify a DOM node.
- \text{style} is the filename or URL of an XSL stylesheet.
- \text{dest} is the filename or URL of the desired output document. If \text{dest} is absent or empty, the function uses a temporary filename. If \text{dest} is '-tostring', the function returns the output document as a MATLAB string.

\[\text{result,style}] = \text{xslt}(\ldots)\] returns a processed stylesheet appropriate for passing to subsequent XSLT calls as \text{style}. This prevents costly repeated processing of the stylesheet.

\text{xslt}(\ldots,'-\text{web}') displays the resulting document in the Help Browser.

**Remarks**
Find out more about XSL stylesheets and how to write them at the World Wide Web Consortium (W3C) web site, [http://www.w3.org/Style/XSL/](http://www.w3.org/Style/XSL/).

**Example**
This example converts the file info.xml using the stylesheet info.xsl, writing the output to the file info.html. It launches the resulting HTML file in the Help Browser. MATLAB has several info.xml files that are used by the Start menu.

\text{xslt info.xml info.xsl info.html -web}
See Also xmlread, xmlwrite
### Purpose
Create array of all zeros

#### Syntax

- \( B = \text{zeros}(n) \)
- \( B = \text{zeros}(m,n) \)
- \( B = \text{zeros}([m,n]) \)
- \( B = \text{zeros}(m,n,p,...) \)
- \( B = \text{zeros}([m,n,p,...]) \)
- \( B = \text{zeros}(\text{size}(A)) \)
- \( \text{zeros}(m,n,...,\text{classname}) \)
- \( \text{zeros}([m,n,...],\text{classname}) \)

#### Description

- \( B = \text{zeros}(n) \) returns an \( n \)-by-\( n \) matrix of zeros. An error message appears if \( n \) is not a scalar.

- \( B = \text{zeros}(m,n) \) or \( B = \text{zeros}([m,n]) \) returns an \( m \)-by-\( n \) matrix of zeros.

- \( B = \text{zeros}(m,n,p,...) \) or \( B = \text{zeros}([m,n,p,...]) \) returns an \( m \)-by-\( n \)-by-\( p \)-by-\( ... \) array of zeros.

#### Note
The size inputs \( m, n, p, ... \) should be nonnegative integers. Negative integers are treated as 0.

- \( B = \text{zeros}(\text{size}(A)) \) returns an array the same size as \( A \) consisting of all zeros.

- \( \text{zeros}(m,n,...,\text{classname}) \) or \( \text{zeros}([m,n,...],\text{classname}) \) is an \( m \)-by-\( n \)-by-\( ... \) array of zeros of data type \( \text{classname} \). \( \text{classname} \) is a string specifying the data type of the output. \( \text{classname} \) can have the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

#### Example

- \( x = \text{zeros}(2,3,'\text{int}8'); \)

#### Remarks

The MATLAB language does not have a dimension statement; MATLAB automatically allocates storage for matrices. Nevertheless, for large
matrices, MATLAB programs may execute faster if the zeros function is used to set aside storage for a matrix whose elements are to be generated one at a time, or a row or column at a time. For example

```matlab
x = zeros(1,n);
for i = 1:n, x(i) = i; end
```

**See Also**

`eye`, `ones`, `rand`, `randn`, `complex`
**Purpose**  
Compress files into zip file

**Syntax**

```matlab
zip(zipfile,files)
zip(zipfile,files,rootdir)
entrynames = zip(...)
```

**Description**

`zip(zipfile,files)` creates a zip file with the name `zipfile` from the list of files and directories specified in `files`. Relative paths are stored in the zip file, but absolute paths are not. Directories recursively include all of their content.

`zipfile` is a string specifying the name of the zip file. If `zipfile` has no extension, MATLAB appends the `.zip` extension.

`files` is a string or cell array of strings containing the list of files or directories included in `zipfile`. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current directory or with an absolute path.
Directories must be specified relative to the current directory or with absolute paths. On UNIX systems, directories can also start with ~ or ~username/, which expands to the current user's home directory or the specified user's home directory, respectively. The wildcard character * can be used when specifying files or directories, except when relying on the MATLAB path to resolve a file name or partial path name.

\( \text{zip(zipfile,files,rootdir)} \) allows the path for files to be specified relative to rootdir rather than the current directory.

\( \text{entrynames = zip(...)} \) returns a string cell array of the names of the files contained in zipfile. If files contains a relative path, entrynames also contains the relative path.

**Examples**

**Zip a File**

Create a zip file of the file guide.viewlet, which is in the MATLAB demos directory. It saves the zip file in d:/mymfiles/viewlet.zip.

```matlab
file = fullfile(matlabroot,'demos','guide.viewlet');
zip('d:/mymfiles/viewlet.zip',file)
```

Run \( \text{zip} \) for the files guide.viewlet and import.viewlet and save the zip file in viewlets.zip. The source files and zipped file are in the current directory.

```matlab
zip('viewlets.zip', {'guide.viewlet', 'import.viewlet'})
```

**Zip Selected Files**

Run \( \text{zip} \) for all .m and .mat files in the current directory to the file backup.zip:

```matlab
zip('backup', {'*.m', '*.mat'})
```

31. UNIX is a registered trademark of The Open Group in the United States and other countries.
**Zip a Directory**

Run `zip` for the directory `D:/mymfiles` and its contents to the zip file `mymfiles` in the directory one level up from the current directory.

```python
zip('..:/mymfiles','D:/mymfiles')
```

**Zip Between Directories**

Run `zip` for the files `thesis.doc` and `defense.ppt`, which are located in `d:/PhD`, to the zip file `thesis.zip` in the current directory.

```python
zip('thesis.zip', {'thesis.doc','defense.ppt'}, 'd:/PhD')
```

**See Also**

`gzip`, `gunzip`, `tar`, `untar`, `unzip`
Purpose

Turn zooming on or off or magnify by factor

GUI

Use the **Zoom** tools on the figure toolbar to zoom in or zoom out on a plot, or select **Zoom In** or **Zoom Out** from the figure’s **Tools** menu. For details, see “Enlarging the View” in the MATLAB Graphics documentation.

Alternatives

**Syntax**

- `zoom on`
- `zoom off`
- `zoom out`
- `zoom reset`
- `zoom`  
- `zoom xon`
- `zoom yon`
- `zoom(factor)`
- `zoom(fig, option)`
- `h = zoom(figure_handle)`

**Description**

`zoom on` turns on interactive zooming. When interactive zooming is enabled in a figure, pressing a mouse button while your cursor is within an axes zooms into the point or out from the point beneath the mouse. Zooming changes the axes limits. When using zoom mode, you

- **Zoom in** by positioning the mouse cursor where you want the center of the plot to be and either
  - Press the mouse button or
  - Rotate the mouse scroll wheel away from you (upward).

- **Zoom out** by positioning the mouse cursor where you want the center of the plot to be and either
  - Simultaneously press **Shift** and the mouse button, or
  - Rotate the mouse scroll wheel toward you (downward).

Each mouse click or scroll wheel click zooms in or out by a factor of 2.
Clicking and dragging over an axes when zooming in is enabled draws a rubberband box. When you release the mouse button, the axes zoom in to the region enclosed by the rubberband box.

Double-clicking over an axes returns the axes to its initial zoom setting in both zoom-in and zoom-out modes.

`zoom off` turns interactive zooming off.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive zoom mode is enabled, will return to this zoom level.

`zoom` toggles the interactive zoom status between off and on (restoring the most recently used zoom tool).

`zoom xon` and `zoom yon` set `zoom on` for the x- and y-axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by 1/factor.

`zoom(fig, option)` Any of the preceding options can be specified on a figure other than the current figure using this syntax.

`h = zoom(figure_handle)` returns a `zoom mode object` for the figure. `figure_handle` for you to customize the mode’s behavior.

**Using Zoom Mode Objects**

Access the following properties of zoom mode objects via `get` and modify some of them using `set`.

- **Enable 'on' | 'off'** — Specifies whether this figure mode is currently enabled on the figure

- **FigureHandle <handle>** — The associated figure handle, a read-only property that cannot be set
• **Motion** 'horizontal' | 'vertical' | 'both' — The type of zooming enabled for the figure

• **Direction** 'in' | 'out' — The direction of the zoom operation

• **RightClickAction** 'InverseZoom' | 'PostContextMenu' — The behavior of a right-click action

  A value of 'InverseZoom' causes a right-click to zoom out. A value of 'PostContextMenu' displays a context menu. This setting persists between MATLAB sessions.

• **UIContextMenu** <handle> — Specifies a custom context menu to be displayed during a right-click action

  This property is ignored if the RightClickAction property has been set to 'on'.

### Zoom Mode Callbacks

You can program the following callbacks for zoom mode operations.

• **ButtonDownFilter** <function_handle> — Function to intercept ButtonDown events

  The application can inhibit the zoom operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

  ```
  function [res] = myfunction(obj,event_obj)
  % obj handle to the object that has been clicked on
  % event_obj struct for event data (empty in this release)
  % res [output] a logical flag to determine whether the zoom
  % operation should take place or the 'ButtonDownFcn'
  % property of the object should take precedence
  ```

• **ActionPreCallback** <function_handle> — Function to execute before zooming

2-4238
Set this callback if you want to execute code when a zoom operation starts. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

```matlab
function myfunction(obj,event_obj)
    % obj handle to the figure that has been clicked on.
    % event_obj object containing struct of event data
```

The event data has the following field.

<table>
<thead>
<tr>
<th>Axes</th>
<th>The handle of the axes that is being zoomed</th>
</tr>
</thead>
</table>

- **ActionPostCallback** `<function_handle>` — Function to execute after zooming

Set this callback if you want to execute code when a zoom operation finishes. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

```matlab
function myfunction(obj,event_obj)
    % obj handle to the figure that has been clicked on
    % event_obj object containing struct of event data (same as the event data of the 'ActionPreCallback' callback)
```

### Zoom Mode Utility Functions

The following functions in zoom mode query and set certain of its properties.

- **flags = isAllowAxesZoom(h,axes)** — Function querying permission to zoom axes

Calling the function `isAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, as input returns a logical array of the
same dimension as the axes handle vector, which indicates whether a zoom operation is permitted on the axes objects.

- **setAllowAxesZoom(h,axes,flag)** — Function to set permission to zoom axes

  Calling the function `setAllowAxesZoom` on the zoom object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a zoom operation on the axes objects.

- **info = getAxesZoomMotion(h,axes)** — Function to get style of zoom operations

  Calling the function `getAxesZoomMotion` on the zoom object, `H`, with a vector of axes handles, `axes`, as input returns a character cell array of the same dimension as the axes handle vector, which indicates the type of zoom operation for each axes. Possible values for the type of operation are 'horizontal', 'vertical', or 'both'.

- **setAxesZoomMotion(h,axes,style)** — Function to set style of zoom operations

  Calling the function `setAxesZoomMotion` on the zoom object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, sets the style of zooming on each axes.

**Examples**

**Example 1 — Entering Zoom Mode**

Plot a graph and turn on Zoom mode:

```matlab
plot(1:10);
zoom on
% zoom in on the plot
```

**Example 2 — Constrained Zoom**

Create zoom mode object and constrain to x-axis zooming:

```matlab
plot(1:10);
    h = zoom;
set(h,'Motion','horizontal','Enable','on');
```
% zoom in on the plot in the horizontal direction.

**Example 3 — Constrained Zoom in Subplots**

Create four axes as subplots and set zoom style differently for each by setting a different property for each axes handle:

```matlab
ax1 = subplot(2,2,1);
plot(1:10);
h = zoom;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesZoom(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesZoomMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesZoomMotion(h,ax4,'vertical');
% Zoom in on the plots.
```

**Example 4 — Coding a ButtonDown Callback**

Create a buttonDown callback for zoom mode objects to trigger. Copy the following code to a new M-file, execute it, and observe zooming behavior:

```matlab
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = zoom;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse click on the line
%```
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
    flag = true;
else
    flag = false;
end

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-buttonDown events for zoom mode objects to trigger. Copy the following code to a new M-file, execute it, and observe zoom behavior:

```matlab
function demo
% Listen to zoom events
plot(1:10);
% Listen to zoom events
h = zoom;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A zoom is about to occur.');
%
function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%f %f].',newLim));
```

Example 6 — Creating a Context Menu for Zoom Mode

Coding a context menu that lets the user to switch to Pan mode by right-clicking:

```matlab
figure;plot(magic(10))
hCMZ = uicontextmenu;
```
hZMenu = uimenu('Parent',hCMZ,'Label','Switch to pan','Callback','p
hZoom = zoom(gcf);
set(hZoom,'UIContextMenu',hCMZ);
zoom('on')

You cannot add items to the built-in zoom context menu, but you can replace it with your own.

**Remarks**

zoom changes the axes limits by a factor of 2 (in or out) each time you press the mouse button while the cursor is within an axes. You can also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

You can create a zoom mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

**Note**  Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure’s callbacks and the operation will not succeed. The one exception to this rule is the figure WindowButtonMotionFcn callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure’s callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different zoom behaviors to different subplot axes via a mode object and then link them using the linkaxes function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

**See Also**

linkaxes, pan, rotate3d

“Object Manipulation” on page 1-107 for related functions
Symbols and Numerics

\& 2-52 2-59
' 2-40
* 2-40
+ 2-40
- 2-40
/ 2-40 :
< 2-50
> 2-50
@ 2-1443 \ 2-40 ^ 2-40 |
| 2-52 2-59 ~ 2-52 2-59
&& 2-59 == 2-50 || 2-59
~= 2-50
1-norm 2-2570 2-3023
2-norm (estimate of) 2-2572

A

abs 2-69
absolute accuracy
   BVP 2-478
   DDE 2-896
   ODE 2-2620
absolute value 2-69
Accelerator
   Uimenu property 2-3898
accumarray 2-70
accuracy
   of linear equation solution 2-688
   of matrix inversion 2-688
acos 2-77
acosd 2-79
acosh 2-80

acot 2-82
acotd 2-84
acoth 2-85
acsc 2-87
acscl 2-89
acsch 2-90
activelegend 2-2810
actxcontrol 2-92
actxcontrollist 2-100
actxcontrolselect 2-101
actxserver 2-106
Adams-Bashforth-Moulton ODE solver 2-2609
addCause, MException method 2-110
addevent 2-114
addframe
   AVI files 2-116
addition (arithmetic operator) 2-40
addlistener 2-118
addOptional method
   of inputParser object 2-120
addParamValue method
   of inputParser object 2-123
addpath 2-126
addpref function 2-128
addprop dynamicprops method 2-129
addproperty 2-130
addRequired method
   of inputParser object 2-132
addressing selected array elements 2-66
addsample 2-135
addsampletocollection 2-137
adddate 2-139
addevent 2-141
adjacency graph 2-1013
airy 2-143
Airy functions
   relationship to modified Bessel functions 2-143
align function 2-145
aligning scattered data
multi-dimensional 2-2492
two-dimensional 2-1585
\texttt{ALim}, Axes property 2-296
\texttt{all} 2-153
\texttt{allchild} function 2-155
allocation of storage (automatic) 2-4231
\texttt{AlphaData}
  image property 2-1768
  surface property 2-3568
  surfaceplot property 2-3591
\texttt{AlphaDataMapping}
  image property 2-1769
  patch property 2-2713
  surface property 2-3569
  surfaceplot property 2-3591
\texttt{AmbientLightColor}, Axes property 2-297
\texttt{AmbientStrength}, Axes property 2-297
\texttt{Patch} property 2-2714
\texttt{Surface} property 2-3570
\texttt{surfaceplot} property 2-3592
call 2-153
\texttt{allocation of storage (automatic)} 2-4231
\texttt{AlphaData}
  image property 2-1768
  surface property 2-3568
  surfaceplot property 2-3591
\texttt{AlphaDataMapping}
  image property 2-1769
  patch property 2-2713
  surface property 2-3569
  surfaceplot property 2-3591
\texttt{AmbientLightColor}, Axes property 2-297
\texttt{AmbientStrength}, Axes property 2-297
\texttt{Patch} property 2-2714
\texttt{Surface} property 2-3570
\texttt{surfaceplot} property 2-3592
\texttt{AMD} 2-161 2-2058
\texttt{analytical partial derivatives (BVP)} 2-479
\texttt{analyzing graphs}
  deleting annotations 2-171
  in plot edit mode 2-2811
\texttt{Annotation}
  areaserises property 2-222
  contourgroup property 2-714
  errorbarseries property 2-1090
  hggroup property 2-1674
  hgtransform property 2-1703
  image property 2-1769
  line property 2-356 2-2117
\texttt{lineseries} property 2-2132
\texttt{Patch} property 2-2714
\texttt{quivergroup} property 2-2972
\texttt{rectangle} property 2-3042
\texttt{scattergroup} property 2-3201
\texttt{stairseries} property 2-3386
\texttt{stemseries} property 2-3420
\texttt{Surface} property 2-3570
\texttt{surfaceplot} property 2-3592
text property 2-3675
\texttt{annotationfunction} 2-169
\texttt{ans} 2-212
\texttt{anti-diagonal} 2-1618
\texttt{any} 2-213
\texttt{arccosecant} 2-87
\texttt{arccosine} 2-77
\texttt{arccotangent} 2-82
\texttt{arcsecant} 2-245
\texttt{arcsine} 2-250
\texttt{arctangent} 2-259
  four-quadrant 2-261
\texttt{arguments, M-file}
  checking number of inputs 2-2483
  checking number of outputs 2-2487
  number of input 2-2485
  number of output 2-2485
  passing variable numbers of 2-4092
\texttt{arithmetic operations, matrix and array}
  distinguished 2-40
\texttt{arithmetic operators}
  reference 2-40
\texttt{array}
  addressing selected elements of 2-66
  dimension
    rearrange 2-1354
  displaying 2-990
  flip dimension of 2-1354
  left division (arithmetic operator) 2-42
  maximum elements of 2-2299
  mean elements of 2-2305
median elements of 2-2308
minimum elements of 2-2385
multiplication (arithmetic operator) 2-41
of all ones 2-2641
of all zeros 2-4231
power (arithmetic operator) 2-42
product of elements 2-2885
rearrange
dimension 2-1354
removing first n singleton dimensions of 2-3274
removing singleton dimensions of 2-3373
reshaping 2-3114
reverse dimension of 2-1354
right division (arithmetic operator) 2-41
shift circularly 2-597
shifting dimensions of 2-3274
size of 2-3288
sorting elements of 2-3310
structure 2-1535 2-3140 2-3260
sum of elements 2-3548
swapping dimensions of 2-1924 2-2785
transpose (arithmetic operator) 2-42
arrayfun 2-238
arrays
detecting empty 2-1940
maximum size of 2-686
opening 2-2642
arrays, structure
field names of 2-1231
arrowhead matrix 2-673
ASCII
delimited files
writing 2-1008
ASCII data
converting sparse matrix after loading from 2-3323
reading 2-1004
reading from disk 2-2183
saving to disk 2-3176
ascii function 2-244
asec 2-245
asecd 2-247
asech 2-248
asin 2-250
asind 2-252
asinh 2-253
aspect ratio of axes 2-816 2-2747
assert 2-255
assignin 2-257
atan 2-259
atan2 2-261
atand 2-263
atanh 2-264
.au files
reading 2-281
writing 2-282
audio
saving in AVI format 2-283
signal conversion 2-2110 2-2466
audiodevinfo 2-266
audioplayer 2-268
audiorecorder 2-274
aufinfo 2-280
auread 2-281
AutoScale
quivergroup property 2-2973
AutoScaleFactor
quivergroup property 2-2973
autoselection of OpenGL 2-1269
auwrite 2-282
average of array elements 2-2305
average, running 2-1320
avi 2-283
avifile 2-283
aviinfo 2-287
aviread 2-289
axes 2-290
editing 2-2811
setting and querying data aspect ratio 2-816
setting and querying limits 2-4202
setting and querying plot box aspect ratio 2-2747

A

Axes
creating 2-290
defining default properties 2-294
fixed-width font 2-313
property descriptions 2-296
axis 2-334
axis crossing. See zero of a function
azimuth (spherical coordinates) 2-3339
azimuth of viewpoint 2-4112

B

BackFaceLighting
Surface property 2-3571
surfaceplot property 2-3594
BackFaceLightingpatch property 2-2716
BackgroundColor
annotation textbox property 2-202
Text property 2-3676
Uitable property 2-3970
BackGroundColor
Uicontrol property 2-3850
badly conditioned 2-3023
balance 2-340
BarLayout
barseries property 2-357
BarWidth
barseries property 2-357
base to decimal conversion 2-376
base two operations
conversion from decimal to binary 2-912
logarithm 2-2205
next power of two 2-2566
base2dec 2-376
Baseline
barseries property 2-357
stem property 2-3421

BaseValue
areaseries property 2-223
barseries property 2-358
stem property 2-3421
beep 2-377

BeingDeleted
areaseries property 2-223
barseries property 2-358
contour property 2-715
errorbar property 2-1091
group property 2-1236 2-1770 2-3678
hggroup property 2-1675
hgtransform property 2-1704
light property 2-2100
line property 2-2118
lineseries property 2-2133
quivergroup property 2-2973
rectangle property 2-3043
scatter property 2-3202
stairseries property 2-3387
stem property 2-3421
surface property 2-3571
surfaceplot property 2-3594
transform property 2-2716
Uipushtool property 2-3934
Uitable property 2-3971
Uitoggletool property 2-4000
Uitoolbar property 2-4013
bench 2-378
benchmark 2-378
Bessel functions
first kind 2-387
modified, first kind 2-384
modified, second kind 2-390
second kind 2-393
Bessel functions, modified
relationship to Airy functions 2-143
Bessel’s equation
(defined) 2-387
modified (defined) 2-384
besseli 2-384
besselj 2-387
besselk 2-390
bessely 2-393
beta 2-397
beta function
  (defined) 2-397
  incomplete (defined) 2-399
  natural logarithm 2-402
betainc 2-399
betaln 2-402
bicg 2-403
bicgstab 2-412
bicgstabl 2-418
BiConjugate Gradients method 2-403
BiConjugate Gradients Stabilized method 2-412
  2-418
big endian formats 2-1373
bin2dec 2-421
binary
  data
    writing to file 2-1455
  files
    reading 2-1403
    mode for opened files 2-1371
binary data
  reading from disk 2-2183
  saving to disk 2-3176
binary function 2-422
binary to decimal conversion 2-421
bisection search 2-1465
bit depth
  querying 2-1791
bit-wise operations
  AND 2-424
  get 2-427
  OR 2-431
  set bit 2-432
  shift 2-433
  XOR 2-435
bitand 2-424
bitcmp 2-425
bitget 2-427
bitmaps
  writing 2-1818
bitmax 2-429
bitor 2-431
bitset 2-432
bitshift 2-433
bitxor 2-435
blanks 2-436
  removing trailing 2-908
blkdiag 2-437
BMP files
  writing 2-1818
bold font
  TeX characters 2-3701
boundary value problems 2-485
box 2-438
Box, Axes property 2-298
braces, curly (special characters) 2-62
brackets (special characters) 2-62
break 2-439
breakpoints
  listing 2-866
  removing 2-853
  resuming execution from 2-856
  setting in M-files 2-870
brighten 2-440
browser
  for help 2-1658
brush 2-442
bsxfun 2-452
bubble plot (scatter function) 2-3196
Buckminster Fuller 2-3645
builtin 2-455
BusyAction
  areaseries property 2-223
  Axes property 2-298
  barseries property 2-358
contour property 2-715
errorbar property 2-1092
Figure property 2-1237
hggroup property 2-1676
hgtransform property 2-1705
Image property 2-1771
Light property 2-2100
line property 2-2119
Line property 2-2133
patch property 2-2716
quivergroup property 2-2974
rectangle property 2-3044
Root property 2-3144
scatter property 2-3203
stairseries property 2-3388
stem property 2-3422
Surface property 2-3571
surfaceplot property 2-3594
Text property 2-3678
Uicontextmenu property 2-3835
Uicontrol property 2-3851
Uimenu property 2-3899
Uitoggletool property 2-4001
Uitoolbar property 2-4013

ButtonDownFcn
area series property 2-224
Axes property 2-299
bargraphs property 2-359
contour property 2-716
errorbar property 2-1092
Figure property 2-1237
hggroup property 2-1676
hgtransform property 2-1705
Image property 2-1771
Light property 2-2101
Line property 2-2119
lineseries property 2-2134
patch property 2-2717

quivergroup property 2-2974
rectangle property 2-3044
Root property 2-3144
scatter property 2-3203
stairseries property 2-3388
stem property 2-3422
Surface property 2-3572
surfaceplot property 2-3595
Text property 2-3679
Uicontrol property 2-3679
Uitable property 2-3972

BVP solver properties
analytical partial derivatives 2-479
error tolerance 2-477
Jacobian matrix 2-479
mesh 2-482
singular BVPs 2-482
solution statistics 2-483
vectorization 2-478
bvp4c 2-456
bvp5c 2-467
bvpget 2-472
bvpinit 2-473
bvpset 2-476
bvpxtend 2-485

C

calendar 2-486
call history 2-2892

CallBack
    Uicontextmenu property 2-3836
    Uicontrol property 2-3852
    Uimenu property 2-3900

CallbackObject, Root property 2-3144
calllib 2-487
callSoapService 2-489
camdolly 2-491
camera
dollying position 2-491
moving camera and target postions 2-491
placng a light at 2-495
positioning to view objects 2-497
rotating around camera target 2-499 2-501
rotating around viewing axis 2-505
setting and querying position 2-502
setting and querying projection type 2-504
setting and querying target 2-506
setting and querying up vector 2-508
setting and querying view angle 2-510
CameraPosition, Axes property 2-300
CameraPositionMode, Axes property 2-300
CameraTarget, Axes property 2-301
CameraTargetMode, Axes property 2-301
CameraUpVector, Axes property 2-301
CameraUpVectorMode, Axes property 2-301
CameraViewAngle, Axes property 2-302
CameraViewAngleMode, Axes property 2-302
camlight 2-495
camlookat 2-497
camorbit 2-499
campan 2-501
campos 2-502
camproj 2-504
camroll 2-505
camtarget 2-506
camup 2-508
camva 2-510
camzoom 2-512
cart2pol 2-516
cart2sph 2-518
Cartesian coordinates 2-516 2-518 2-2823 2-3339
case 2-519
    in switch statement (defined) 2-3632
        lower to upper 2-4054
            upper to lower 2-2217
    cast 2-521
cat 2-522
catch 2-524
caxis 2-527
Cayley-Hamilton theorem 2-2843
cd 2-532
cd (ftp) function 2-537
CData
    Image property 2-1772
    scatter property 2-3204
    Surface property 2-3573
    surfaceplot property 2-3596
    Uicontrol property 2-3853
    Uipushtool property 2-3935
    Uitoggletool property 2-4001
CDataMapping
    Image property 2-1774
    patch property 2-2719
    Surface property 2-3574
    surfaceplot property 2-3596
    surfaceplot property 2-3597
CDataMode
    scatter property 2-3204
    surfaceplot property 2-3597
cdf2rdf 2-538
cdfepoch 2-540
cdfinfo 2-541
cdfread 2-545
cdfwrite 2-549
ceil 2-552
cell 2-553
cell array
    conversion to from numeric array 2-2580
    creating 2-553
    structure of, displaying 2-566
cell2mat 2-555
cell2struct 2-557
celledisp 2-559
CellEditCallback
    Uitable property 2-3973
cellfun 2-560
cellplot 2-566
CellSelectionCallback
    Uitable property 2-3975

cgs 2-569
char 2-574
characters
    conversion, in format specification string 2-1392 2-3362
    escape, in format specification string 2-1393 2-3362
check boxes 2-3843
Checked, Uimenu property 2-3900
checkerboard pattern (example) 2-3103
checkin 2-575
    examples 2-576
    options 2-575
checkout 2-578
    examples 2-579
    options 2-578
child functions 2-2887
Children
    areaseries property 2-225
    Axes property 2-303
    barseries property 2-360
    contour property 2-716
    errorbar property 2-1093
    Figure property 2-1238
    hggroup property 2-1677
    hgtransform property 2-1706
    Image property 2-1774
    Light property 2-2101
    Line property 2-2120
    lineseries property 2-2134
    patch property 2-2720
    quivergroup property 2-2975
    rectangle property 2-3045
    Root property 2-3144
    scatter property 2-3205
    stairseries property 2-3389
    stem property 2-3423
    Surface property 2-3574
    surfaceplot property 2-3598
    Text property 2-3680
    Uicontextmenu property 2-3836
    Uicontrol property 2-3854
    Uimenu property 2-3901
    Uitable property 2-3975
    Uitoolbar property 2-4014
chol 2-581
    Cholesky factorization 2-581
        (as algorithm for solving linear equations) 2-2409
        lower triangular factor 2-2701
        preordering for 2-673
cholinc 2-586
cholupdate 2-594
circle
    rectangle function 2-3037
circshift 2-597
cla 2-601
clabel 2-602
class 2-608
    class, object. See object classes
    classes
        field names 2-1231
        loaded 2-1850
clc 2-613 2-623 2-3273
clear 2-614
        serial port I/O 2-622
clearing
        Command Window 2-613
        items from workspace 2-614
        Java import list 2-616
clf 2-623
ClickedCallback
    Uipushtool property 2-3936
    Uitoggletool property 2-4002
CLim, Axes property 2-304
CLimMode, Axes property 2-304
clipboard 2-624
Clipping
areaseries property 2-225
Axes property 2-305
barseries property 2-360
contour property 2-717
errobar property 2-1093
Figure property 2-1239
hggroup property 2-1677
hgtransform property 2-1706
Image property 2-1775
Light property 2-2101
Line property 2-2120
lineseries property 2-2135
quivergroup property 2-2975
rectangle property 2-3045
Root property 2-3145
scatter property 2-3205
stairseries property 2-3389
stem property 2-3423
Surface property 2-3575
surfaceplot property 2-3598
Text property 2-3680
Uicontrol property 2-3854
Uitable property 2-3975
Clippingpatch property 2-2720
clock 2-625
close 2-626
    AVI files 2-628
close (ftp) function 2-629
CloseRequestFcn, Figure property 2-1239
closest point search 2-1032
closest triangle search 2-3795
closing
    files 2-1188
    MATLAB 2-2962
cmapeditor 2-653
cmopts 2-631
cmpermute 2-633
cmunique 2-634
code
    analyzer 2-2413
colamd 2-637
colon operator 2-66
color
    quantization performed by rgb2ind 2-3128
Color
    annotation arrow property 2-173
    annotation doublearrow property 2-177
    annotation line property 2-185
    annotation textbox property 2-202
    Axes property 2-305
    errorbar property 2-1093
    Figure property 2-1241
    Light property 2-2101
    Line property 2-2121
    lineseries property 2-2135
    quivergroup property 2-2976
    stairseries property 2-3389
    stem property 2-3424
    Text property 2-3680
    textarrow property 2-191
color approximation
    performed by rgb2ind 2-3128
color of fonts, see also FontColor property 2-3701
colorbar 2-641
colormap 2-648
editor 2-653
Colormap, Figure property 2-1242
colormaps
    converting from RGB to HSV 2-3126
    plotting RGB components 2-3130
    rearranging colors in 2-633
    removing duplicate entries in 2-634
ColorOrder, Axes property 2-305
ColorSpec 2-671
colperm 2-673
ColumnEditable
    Uitable property 2-3976
ColumnFormat
    Uitable property 2-3976
ColumnName
Uitable property 2-3982

ColumnWidth
Uitable property 2-3982

COM

object methods
actxcontrol 2-92
actxcontrolset 2-100
actxcontrolselect 2-101
actxserver 2-106
addproperty 2-130
delete 2-944
deleteproperty 2-951
eventlisteners 2-1126
events 2-1130
get 2-1511
inspect 2-1866
invoke 2-1921
iscom 2-1936
isevent 2-1949
isinterface 2-1961
ismethod 2-1972
load 2-2188
move 2-2444
propedit 2-2896
registerevent 2-3088
release 2-3093
save 2-3184
set 2-3242
unregisterallevents 2-4038
unregisterevent 2-4041

object properties
isprop 2-1995

server methods
Execute 2-1132
Feval 2-1201

combinations of n elements 2-2491
combs 2-2491
comet 2-675
comet3 2-677
c comma (special characters) 2-64

command syntax 2-1655 2-3650
Command Window
  clearing 2-613
  cursor position 2-1726
  get width 2-680
commandhistory 2-679
commands
  help for 2-1654 2-1664
  system 2-3653
  UNIX 2-4029
commandwindow 2-680
comments
  block of 2-64
common elements. See set operations, intersection
companion matrix 2-681
companion matrix 2-681
compass 2-682
CompilerConfiguration 2-2369
CompilerConfigurationDetails 2-2369
complementary error function
  (defined) 2-1079
  scaled (defined) 2-1079
complete elliptic integral
  (defined) 2-1062
  modulus of 2-1062 2-1062
complex 2-684 2-1759
  exponential (defined) 2-1140
  logarithm 2-2202 to 2-2203
  numbers 2-1735
  numbers, sorting 2-3310 2-3314
  phase angle 2-168
  sine 2-3282
  unitary matrix 2-2925
  See also imaginary
complex conjugate 2-698
  sorting pairs of 2-775
complex data
  creating 2-684
  complex numbers, magnitude 2-69
complex Schur form 2-3219
compression
  lossy 2-1822
computer 2-686
computer MATLAB is running on 2-686
concatenation
  of arrays 2-522
cond 2-688
condeig 2-689
condest 2-690
condition number of matrix 2-688 2-3023
  improving 2-340
coneplot 2-692
conj 2-698
conjugate, complex 2-698
  sorting pairs of 2-775
connecting to FTP server 2-1435
containers
  Map 2-1963 2-2042 2-2081 2-2254 2-3098
  2-3291 2-4085
contents.m file 2-1655
context menu 2-3831
continuation (... , special characters) 2-64
continue 2-699
continued fraction expansion 2-3017
contour
  and mesh plot 2-1160
  filled plot 2-1152
  functions 2-1148
  of mathematical expression 2-1149
  with surface plot 2-1178
contour3 2-705
contourc 2-708
contourf 2-710
ContourMatrix
  contour property 2-717
contours
  in slice planes 2-735
contourslice 2-735
contrast 2-739
conv 2-740
conv2 2-742
conversion
  base to decimal 2-376
  binary to decimal 2-421
  Cartesian to cylindrical 2-516
  Cartesian to polar 2-516
  complex diagonal to real block diagonal 2-538
  cylindrical to Cartesian 2-2823
  decimal number to base 2-905 2-911
  decimal to binary 2-912
  decimal to hexadecimal 2-913
  full to sparse 2-3320
  hexadecimal to decimal 2-1668
  integer to string 2-1880
  lowercase to uppercase 2-4054
  matrix to string 2-2264
  numeric array to cell array 2-2580
  numeric array to logical array 2-2206
  numeric array to string 2-2584
  partial fraction expansion to
    pole-residue 2-3116
    polar to Cartesian 2-2823
    pole-residue to partial fraction
      expansion 2-3116
    real to complex Schur form 2-3173
  spherical to Cartesian 2-3339
  string matrix to cell array 2-568
  string to numeric array 2-3447
  uppercase to lowercase 2-2217
  vector to character string 2-574
conversion characters in format specification
  string 2-1392 2-3362
convex hulls
  multidimensional visualization 2-751
  two-dimensional visualization 2-748
convhull 2-748
convhulln 2-751
convn 2-754
convolution 2-740
inverse. See deconvolution

two-dimensional 2-742
coordinate system and viewpoint 2-4112
coordinates
  Cartesian 2-516 2-518 2-2823 2-3339
cylindrical 2-516 2-518 2-2823
polar 2-516 2-518 2-2823
spherical 2-3339
coordinates. 2-516
  See also conversion
copyfile 2-755
copyobj 2-758
corrcoef 2-760
cos 2-763
cosd 2-765
cosecant
  hyperbolic 2-789
  inverse 2-87
  inverse hyperbolic 2-90
cosh 2-766
cosine 2-763
  hyperbolic 2-766
  inverse 2-77
  inverse hyperbolic 2-80
cot 2-768
cotangent 2-768
  hyperbolic 2-771
  inverse 2-82
  inverse hyperbolic 2-85
cotd 2-770
coth 2-771
cov 2-773
cplxpair 2-775
cputime 2-776
create, RandStream method 2-777
createClassFromWsd1 2-779
createCopy method
  of inputParser object 2-781
CreateFcn
  areareaseries property 2-225

Axes property 2-306
barseries property 2-360
contour property 2-718
errorbar property 2-1094
Figure property 2-1242
group property 2-1706
hgroup property 2-1677
Image property 2-1775
Light property 2-2102
Line property 2-2121
lineseries property 2-2135
patch property 2-2720
quivergroup property 2-2976
rectangle property 2-3046
Root property 2-3145
scatter property 2-3205
stairseries property 2-3390
stemseries property 2-3424
Surface property 2-3575
surfaceplot property 2-3598
Text property 2-3680
Uicontextmenu property 2-3836
Uicontrol property 2-3854
Uimenu property 2-3901
Uipushtool property 2-3936
Uitable property 2-3983
Uitoggletool property 2-4002
Uitoolbar property 2-4014
createSoapMessage 2-783
creating your own MATLAB functions 2-1441
cross 2-785
cross product 2-785
csc 2-786
cscd 2-788
csch 2-789
csvread 2-791
csvwrite 2-794
ctranspose (M-file function equivalent for \q) 2-46
ctranspose (timeseries) 2-796
cubic interpolation 2-1897 2-1900 2-1903 2-2757
    piecewise Hermite 2-1887
cubic spline interpolation
    one-dimensional 2-1887 2-1897 2-1900 2-1903
cumprod 2-798
cumsum 2-800
cumtrapz 2-802
cumulative
    product 2-798
    sum 2-800
curl 2-804
curly braces (special characters) 2-62
current directory 2-2918
    changing 2-532
CurrentAxes 2-1243
CurrentAxes, Figure property 2-1243
CurrentCharacter, Figure property 2-1244
CurrentFigure, Root property 2-3145
CurrentObject, Figure property 2-1244
CurrentPoint
    Axes property 2-306
    Figure property 2-1244
cursor images
    reading 2-1805
cursor position 2-1726
Curvature, rectangle property 2-3047
curve fitting (polynomial) 2-2835
customverctrl 2-807
Cuthill-McKee ordering, reverse 2-3635 2-3645
cylinder 2-808
cylindrical coordinates 2-516 2-518 2-2823

D

daqread 2-811
daspect 2-816
data
    ASCII
        reading from disk 2-2183
    binary
        writing to file 2-1455
    binary, saving to disk 2-3176
    computing 2-D stream lines 2-3457
    computing 3-D stream lines 2-3459
    formatted
        reading from files 2-1421
        writing to file 2-1391
    formatting 2-1391 2-3360
    isosurface from volume data 2-1987
    reading binary from disk 2-2183
    reading from files 2-3706
    reducing number of elements in 2-3062
    smoothing 3-D 2-3303
    writing to strings 2-3360
Data
    Uitable property 2-3984
data aspect ratio of axes 2-816
data brushing
    different plot types 2-443
    gestures for 2-448
    restrictions on 2-445
data types
    complex 2-684
data, aligning scattered
    multi-dimensional 2-2492
    two-dimensional 2-1585
data, ASCII
    converting sparse matrix after loading
        from 2-3323
DataAspectRatio, Axes property 2-308
DataAspectRatioMode, Axes property 2-311
datatipinfo 2-827
date 2-828
date and time functions 2-1073
date string
    format of 2-833
date vector 2-850
datenum 2-829
Index

datestr 2-833
datevec 2-849
dbclear 2-853
dbcont 2-856
dbdowm 2-857
dblquad 2-858
dbmx 2-860
dbquit 2-861
dbstack 2-863
dbstatus 2-866
dbstep 2-868
dbstop 2-870
dbtype 2-881
dbup 2-882
DDE solver properties
  error tolerance 2-895
  event location 2-901
  solver output 2-897
  step size 2-899
dde23 2-883
ddeget 2-888
ddephas2 output function 2-898
ddephas3 output function 2-898
ddeplot output function 2-898
ddeprint output function 2-898
ddesd 2-889
ddeset 2-894
deal 2-905
deblank 2-908
debugging
  changing workspace context 2-857
  changing workspace to calling M-file 2-882
  displaying function call stack 2-863
  M-files 2-2041 2-2887
  MEX-files on UNIX 2-860
  removing breakpoints 2-853
  resuming execution from breakpoint 2-868
  setting breakpoints in 2-870
  stepping through lines 2-868
dec2base 2-905 2-911
dec2bin 2-912
dec2hex 2-913
decic function 2-914
decimal number to base conversion 2-905 2-911
decimal point (.)
  (special characters) 2-63
  to distinguish matrix and array operations 2-40
development
  Dulmage-Mendelsohn 2-1012
  "economy-size" 2-2925 2-3624
  orthogonal-triangular (QR) 2-2925
  Schur 2-3219
  singular value 2-3016 2-3624
decconv 2-916
decovolution 2-916
definite integral 2-2937
del operator 2-917
del2 2-917
delaunay 2-926
Delaunay tessellation
  3-dimensional visualization 2-933
  multidimensional visualization 2-937
Delaunay triangulation
  visualization 2-926
delaunay3 2-933
delaunayn 2-937
delete 2-942 2-944
  serial port I/O 2-948
  timer object 2-950
delete (ftp) function 2-946
delete handle method 2-947
DeleteFcn
  areaversies property 2-226
  Axes property 2-312
  barseries property 2-361
  contour property 2-718
  errorbar property 2-1094
  Figure property 2-1246
  hgggroup property 2-1678
Index

hgtransform property 2-1707
Image property 2-1775
Light property 2-2103
lineseries property 2-2136
quivergroup property 2-2976
Root property 2-3145
scatter property 2-3206
stairseries property 2-3390
stem property 2-3425
Surface property 2-3575
surfaceplot property 2-3599
Text property 2-3684
Uicontextmenu property 2-3837 2-3855
Uimenu property 2-3902
Uipushtool property 2-3937
Uitable property 2-3985
Uitoggletool property 2-4003
Uitoolbar property 2-4016
DeleteFcn, line property 2-2122
DeleteFcn, rectangle property 2-3047
DeleteFcnpatch property 2-2721
deleteproperty 2-951
deleting
files 2-942
items from workspace 2-614
delevent 2-953
delimitters in ASCII files 2-1004 2-1008
delsample 2-954
delsamplefromcollection 2-955
demo 2-956
demos
in Command Window 2-1036
density
of sparse matrix 2-2567
depdir 2-962
dependence, linear 2-3540
dependent functions 2-2887
depfun 2-963
derivative
approximate 2-979
det 2-967
determinant of a matrix 2-967
detrend 2-968
detrend (timeseries) 2-970
deval 2-971
diag 2-973
diagonal 2-973
anti- 2-1618
k-th (illustration) 2-3766
main 2-973
sparse 2-3325
dialog 2-975
diary 2-977
Diary, Root property 2-3146
DiaryFile, Root property 2-3146
diff 2-979
differences
between adjacent array elements 2-979
between sets 2-3256
differential equation solvers
defining an ODE problem 2-2611
ODE boundary value problems 2-456 2-467
adjusting parameters 2-476
extracting properties 2-472
extracting properties of 2-1112 to 2-1113
2-3763 to 2-3764
forming initial guess 2-473
ODE initial value problems 2-2598
adjusting parameters of 2-2618
extracting properties of 2-2617
parabolic-elliptic PDE problems 2-2766
diffuse 2-981
DiffuseStrength
Surface property 2-3576
surfaceplot property 2-3599
DiffuseStrengthpatch property 2-2721
digamma function 2-2900
dimension statement (lack of in MATLAB) 2-4231
dimensions
size of 2-3288
Diophantine equations 2-1496
dir 2-982
dir (ftp) function 2-987
directive term of a partial fraction expansion 2-3116
directive
%#eml 2-2416
%#ok 2-2416
directories 2-532
adding to search path 2-126
checking existence of 2-1135
copying 2-755
creating 2-2396
listing 2-2218
listing contents of 2-982
listing MATLAB files in 2-4167
removing 2-3136
removing from search path 2-3141
See also directory, search path
directory 2-982
changing on FTP server 2-537
listing for FTP server 2-987
making on FTP server 2-2399
MATLAB location 2-2274
root 2-2274
temporary system 2-3663
See also directories
directory, changing 2-532
directory, current 2-2918
disconnect 2-629
discontinuities, eliminating (in arrays of phase angles) 2-4050
discontinuities, plotting functions with 2-1176
discontinuous problems 2-1369
disp 2-990
memmapfile object 2-992
serial port I/O 2-995
timer object 2-996
disp, MException method 2-993
display 2-998
display format 2-1379
displaying output in Command Window 2-2442
DisplayName
area series property 2-226
bar series property 2-361
contour group property 2-719
error bar series property 2-1094
hg group property 2-1678
hg transform property 2-1708
image property 2-1776
Line property 2-2123
line series property 2-2136
Patch property 2-2721
quiver group property 2-2977
rectangle property 2-3048
scatter group property 2-3206
stair series property 2-3391
Index-16
stemseries property 2-3425
surface property 2-3577
surfaceplot property 2-3600
text property 2-3682
distribution
  Gaussian 2-1079
dither 2-1000
division
  array, left (arithmetic operator) 2-42
  array, right (arithmetic operator) 2-41
  by zero 2-1842
  matrix, left (arithmetic operator) 2-41
  matrix, right (arithmetic operator) 2-41
  of polynomials 2-916
divisor
  greatest common 2-1496
dll libraries
  MATLAB functions
    calllib 2-487
    libfunctions 2-2085
    libfunctionsview 2-2086
    libisloaded 2-2087
    libpointer 2-2089
    libstruct 2-2091
    loadlibrary 2-2192
    unloadlibrary 2-4032
dlmread 2-1004
dlmwrite 2-1008
dmperm 2-1012
Dockable, Figure property 2-1247
docsearch 2-1020
documentation
  displaying online 2-1658
dolly camera 2-491
dos 2-1023
  UNC pathname error 2-1024
dot 2-1025
dot product 2-785 2-1025
dot-parentheses (special characters 2-64
  double 2-1026
double click, detecting 2-1271
double integral
  numerical evaluation 2-858
DoubleBuffer, Figure property 2-1247
downloading files from FTP server 2-2384
dragrect 2-1027
drawing shapes
  circles and rectangles 2-3037
DrawMode, Axes property 2-312
drawnow 2-1029
dsearch 2-1031
dsearchn 2-1032
Dulmage-Mendelsohn decomposition 2-1012
dynamic fields 2-64
dynamicprops class 2-1033
dynamicprops.addprop 2-129

echo 2-1034
Echo, Root property 2-3146
echodemo 2-1036
edge finding, Sobel technique 2-744
EdgeAlpha
  patch property 2-2722
  surface property 2-3577
  surfaceplot property 2-3600
EdgeColor
  annotation ellipse property 2-182
  annotation rectangle property 2-188
  annotation textbox property 2-202
  areaseries property 2-227
  barseries property 2-362
  patch property 2-2723
  Surface property 2-3578
  surfaceplot property 2-3601
  Text property 2-3683
EdgeColor, rectangle property 2-3049
EdgeLighting
  patch property 2-2723
Surface property 2-3579
surfaceplot property 2-3602
editable text 2-3843
editing
  M-files 2-1041
eig 2-1044
eigensystem
  transforming 2-538
eigenvalue
  accuracy of 2-1044
  complex 2-538
  matrix logarithm and 2-2211
  modern approach to computation of 2-2828
  of companion matrix 2-681
  problem 2-1045 2-2833
  problem, generalized 2-1045 2-2833
  problem, polynomial 2-2833
  repeated 2-1046
  Wilkinson test matrix and 2-4188
eigenvalues
  effect of roundoff error 2-340
  improving accuracy 2-340
eigenvector
  left 2-1045
  matrix, generalized 2-2993
  right 2-1045
eigs 2-1050
elevation (spherical coordinates) 2-3339
elevation of viewpoint 2-4112
ellipj 2-1060
ellipke 2-1062
ellipsoid 2-1064
elliptic functions, Jacobian
  (defined) 2-1060
elliptic integral
  complete (defined) 2-1062
  modulus of 2-1060 2-1062
else 2-1066
elseif 2-1067
  %#eml 2-2416
Enable
  Uicontrol property 2-3856
  Uimenu property 2-3903
  Uipushtool property 2-3938
  Uitable property 2-3985
  Uitogglehtool property 2-4004
end 2-1071
end caps for isosurfaces 2-1977
end of line, indicating 2-64
end-of-file indicator 2-1196
eomday 2-1073
eps 2-1074
eq 2-1076
eq, MException method 2-1078
equal arrays
  detecting 2-1943 2-1947
equal sign (special characters) 2-63
equations, linear
  accuracy of solution 2-688
EraseMode
  areaseries property 2-227
  barseries property 2-362
  contour property 2-719
  errorbar property 2-1095
  hggroup property 2-1679
  hgtransform property 2-1708
  Image property 2-1777
  Line property 2-2124
  lineseries property 2-2137
  quivergroup property 2-2978
  rectangle property 2-3049
  scatter property 2-3207
  stairseries property 2-3392
  stem property 2-3426
  Surface property 2-3579
  surfaceplot property 2-3602
  Text property 2-3684
EraseModepatch property 2-2724
error 2-1081
  roundoff. See roundoff error
error function
  complementary 2-1079
  (defined) 2-1079
  scaled complementary 2-1079
error message
  displaying 2-1081
  Index into matrix is negative or zero 2-2207
  retrieving last generated 2-2047 2-2055
error messages
  Out of memory 2-2678
error tolerance
  BVP problems 2-477
  DDE problems 2-895
  ODE problems 2-2619
errorbars, confidence interval 2-1086
errordlg 2-1108
ErrorMessage, Root property 2-3146
errors
  in file input/output 2-1198
  MException class 2-1078
    addCause 2-110
    constructor 2-2375
    disp 2-993
    eq 2-1078
    getReport 2-1549
    isequal 2-1946
    last 2-2045
    ne 2-2499
    rethrow 2-3123
    throw 2-3730
    throwAsCaller 2-3733
Errortype, Root property 2-3147
escape characters in format specification
  string 2-1393 2-3362
etime 2-1111
etree 2-1112
etreeplot 2-1113
eval 2-1114
evalc 2-1117
evalin 2-1118

event location (DDE) 2-901
event location (ODE) 2-2626
event.EventData 2-1120
event.listener 2-1122
event.PropertyEvent 2-1121
event.proplistener 2-1124
eventlisteners 2-1126
events 2-1129 to 2-1130
examples
  calculating isosurface normals 2-1984
  contouring mathematical expressions 2-1149
  isosurface end caps 2-1977
  isosurfaces 2-1988
  mesh plot of mathematical function 2-1158
  mesh/contour plot 2-1162
  plotting filled contours 2-1153
  plotting function of two variables 2-1166
  plotting parametric curves 2-1169
  polar plot of function 2-1172
  reducing number of patch faces 2-3059
  reducing volume data 2-3062
  subsampling volume data 2-3545
  surface plot of mathematical function 2-1176
  surface/contour plot 2-1180
Excel spreadsheets
  loading 2-4207
exclamation point (special characters) 2-65
Execute 2-1132
executing statements repeatedly 2-1377 2-4175
executing statements repeatedly in parallel 2-2694
execution
  improving speed of by setting aside storage 2-4231
  pausing M-file 2-2745
  resuming from breakpoint 2-856
  time for M-files 2-2887
exifread 2-1134
exist 2-1135
exit 2-1139
exp 2-1140
expint 2-1141
expm 2-1142
expm1 2-1144
exponential 2-1140
  complex (defined) 2-1140
  integral 2-1141
  matrix 2-1142
exponentiation
  array (arithmetic operator) 2-42
  matrix (arithmetic operator) 2-42
export2wsdlg 2-1145
extension, filename
  .m 2-1441
  .mat 2-3176
Extent
  Text property 2-3686
  Uicontrol property 2-3856
  Uitable property 2-3986
eye 2-1147
ezcontour 2-1148
ezcontourf 2-1152
ezmesh 2-1156
ezmeshc 2-1160
ezplot 2-1164
ezplot3 2-1168
ezpolar 2-1171
ezsurf 2-1174
ezsurfc 2-1178
annotation rectangle property 2-188
areaseries property 2-229
barseries property 2-364
Surface property 2-3581
surfaceplot property 2-3604
FaceColor, rectangle property 2-3050
FaceColorpatch property 2-2726
FaceLighting
  Surface property 2-3581
  surfaceplot property 2-3605
FaceLightingpatch property 2-2726
faces, reducing number in patches 2-3058
Faces, patch property 2-2727
FaceVertexAlphaData, patch property 2-2728
FaceVertexCDATA, patch property 2-2728
factor 2-1185
factorial 2-1186
factorization 2-2925
  LU 2-2234
  QZ 2-2834 2-2993
  See also decomposition
factorization, Cholesky 2-581
  (as algorithm for solving linear
  equations) 2-2409
  preordering for 2-673
factors, prime 2-1185
false 2-1187
fclose 2-1188
  serial port I/O 2-1189
feather 2-1191
feof 2-1196
ferror 2-1198
feval 2-1199
feval 2-1201
fft 2-1206
FFT. See Fourier transform
fft2 2-1211
fftn 2-1212
fftshift 2-1214
fftw 2-1217
FFTW 2-1209
fgetl 2-1222
  serial port I/O 2-1224
fgets 2-1227
  serial port I/O 2-1228
field names of a structure, obtaining 2-1231
fieldnames 2-1231
fields, noncontiguous, inserting data into 2-1455
fields, of structures
  dynamic 2-64
figure 2-1233
Figure
  creating 2-1233
  defining default properties 2-1235
  properties 2-1236
  redrawing 2-3065
figure windows
  moving in front of MATLAB® desktop 2-3273
figure windows, displaying 2-1333
figurepalette 2-1291
figures
  annotating 2-2811
  opening 2-2642
  saving 2-3188
Figures
  updating from M-file 2-1029
file
  extension, getting 2-1308
  modification date 2-982
  position indicator
    finding 2-1310 2-1434
    setting 2-1432
    setting to start of file 2-1420
file formats
  getting list of supported formats 2-1794
  reading 2-811 2-1803
  writing 2-1816
file size
  querying 2-1791
fileattrib 2-1293
filebrowser 2-1300
filemarker 2-1306
filename
  building from parts 2-1438
  parts 2-1308
  temporary 2-3664
filename extension
  .m 2-1441
  .mat 2-3176
fileparts 2-1308
fileread 2-1310
files 2-1188
  ASCII delimited
    reading 2-1004
    writing 2-1008
  beginning of, rewinding to 2-1420 2-1800
  checking existence of 2-1135
  closing 2-1188
  contents, listing 2-3803
  copying 2-755
  deleting 2-942
  deleting on FTP server 2-946
  end of, testing for 2-1196
  errors in input or output 2-1198
  Excel spreadsheets
    loading 2-4207
  fig 2-3188
  figure, saving 2-3188
  finding position within 2-1310 2-1434
  getting next line 2-1222
  getting next line (with line terminator) 2-1227
  listing
    in directory 2-4167
    names in a directory 2-982
  listing contents of 2-3803
  locating 2-4172
mdl 2-3188
model, saving 2-3188
opening 2-1370 2-2642
opening in Windows applications 2-4189
path, getting 2-1308
pathname for 2-4172
reading
  binary 2-1403
  data from 2-3706
  formatted 2-1421
reading data from 2-811
reading image data from 2-1803
rewinding to beginning of 2-1420 2-1800
setting position within 2-1432
size, determining 2-985
sound
  reading 2-281 2-4158
  writing 2-282 to 2-283 2-4163
startup 2-2273
version, getting 2-1308
.wav
  reading 2-4158
  writing 2-4163
WK1
  loading 2-4193
  writing to 2-4195
writing binary data to 2-1455
writing formatted data to 2-1391
writing image data to 2-1816
See also file
filespec 2-1311
fill 2-1313
Fill
  contour property 2-721
fill3 2-1316
filter 2-1319
  digital 2-1319
  finite impulse response (FIR) 2-1319
  infinite impulse response (IIR) 2-1319
  two-dimensional 2-742
filter (timeseries) 2-1322
filter2 2-1325
find 2-1327
findall function 2-1332
findfigs 2-1333
finding 2-1327
  sign of array elements 2-3281
  zero of a function 2-1461
See also detecting
findobj 2-1334
findobj handle method 2-1337
findprop handle method 2-1338
findstr 2-1339
finish 2-1340
finish.m 2-2962
FIR filter 2-1319
FitBoxToText, annotation textbox property 2-203
FitHeightToText
  annotation textbox property 2-203
fitsinfo 2-1342
fitsread 2-1351
fix 2-1353
fixed-width font
  axes 2-313
  text 2-3687
  uicontrols 2-3858
  uitables 2-3987
FixedColors, Figure property 2-1248
FixedWidthFontName, Root property 2-3146
flints 2-2466
flip
  array dimension 2-1354
flip array
  along dimension 2-1354
flip matrix
  on horizontal axis 2-1356
  on vertical axis 2-1355
flipdim 2-1354
fliplr 2-1355
flipud 2-1356
floating-point
integer, maximum 2-429
floating-point arithmetic, IEEE
  smallest positive number 2-3032
floor 2-1358
flow control
  break 2-439
  case 2-519
  end 2-1071
  error 2-1082
  for 2-1377
  keyboard 2-2041
  otherwise 2-2677
  parfor 2-2694
  return 2-3125
  switch 2-3632
  while 2-4175
fminbnd 2-1360
fminsearch 2-1365
font
  fixed-width, axes 2-313
  fixed-width, text 2-3687
  fixed-width, uitables 2-3858
  fixed-width, uicontrols 2-3857
FontAngle
  annotation textbox property 2-205
  Axes property 2-313
  Text property 2-3686
  Uicontrol property 2-3857
  Uitable property 2-3987
FontName
  annotation textbox property 2-205
  Axes property 2-313
  Text property 2-3686
  textarrow property 2-192
  Uicontrol property 2-3857
  Uitable property 2-3987
fonts
  bold 2-192 2-206 2-3687
  italic 2-192 2-205 2-3686
  specifying size 2-3687
TeX characters
  bold 2-3701
  italics 2-3701
  specifying family 2-3701
  specifying size 2-3701
  units 2-192 2-206 2-3687
FontSize
  annotation textbox property 2-206
  Axes property 2-314
  Text property 2-3687
  textarrow property 2-192
  Uicontrol property 2-3858
  Uitable property 2-3988
FontUnits
  Axes property 2-314
  Text property 2-3687
  Uicontrol property 2-3859
  Uitable property 2-3988
FontWeight
  annotation textbox property 2-206
  Axes property 2-314
  Text property 2-3687
  textarrow property 2-192
  Uicontrol property 2-3859
  Uitable property 2-3988
fopen 2-1370
  serial port I/O 2-1374
for 2-1376
ForegroundColor
  Uicontrol property 2-3859
  Uimenu property 2-3903
  Uitable property 2-3988
format 2-1379
  precision when writing 2-1403
  reading files 2-1422
  specification string, matching file data
to 2-3377
Format 2-3147
formats
  big endian 2-1373
little endian 2-1373
FormatSpacing, Root property 2-3148
formatted data
   reading from file 2-1421
   writing to file 2-1391
formatting data 2-3360
Fourier transform
   algorithm, optimal performance of 2-1209
      2-1745 2-1747 2-2566
   as method of interpolation 2-1902
   convolution theorem and 2-741
   discrete, n-dimensional 2-1212
   discrete, one-dimensional 2-1206
   discrete, two-dimensional 2-1211
   fast 2-1206
   inverse, n-dimensional 2-1749
   inverse, one-dimensional 2-1745
   inverse, two-dimensional 2-1747
   shifting the zero-frequency component of 2-1215
fplot 2-1386 2-1402
fprintf 2-1391
   displaying hyperlinks with 2-1396
   serial port I/O 2-1398
fraction, continued 2-3017
fragmented memory 2-2678
frame2im 2-1402
frames 2-3843
fread 2-1403
   serial port I/O 2-1413
freqspace 2-1419
frequency response
   desired response matrix
   frequency spacing 2-1419
frequency vector 2-2214
frewind 2-1420
fromName meta.class method 2-2339
fromName meta.package method 2-2350
fscanf 2-1421
   serial port I/O 2-1428
fseek 2-1432
ftell 2-1434
FTP
   connecting to server 2-1435
ftp function 2-1435
full 2-1437
fullfile 2-1438
func2str 2-1439
function 2-1441
function handle 2-1443
function handles
   overview of 2-1443
function syntax 2-1655 2-3650
functions 2-1446
   call history 2-2892
   call stack for 2-863
   checking existence of 2-1135
   clearing from workspace 2-614
   finding using keywords 2-2215
   help for 2-1654 2-1664
   in memory 2-1850
   locating 2-4172
   pathname for 2-4172
   that work down the first non-singleton dimension 2-3274
funm 2-1450
fwrite 2-1455
   serial port I/O 2-1457
fzero 2-1461

G

gallery 2-1467
gamma function
   (defined) 2-1490
   incomplete 2-1490
   logarithm of 2-1490
   logarithmic derivative 2-2900
   Gauss-Kronrod quadrature 2-2951
   Gaussian distribution function 2-1079
Gaussian elimination
(as algorithm for solving linear equations) 2-1917 2-2410
Gauss Jordan elimination with partial pivoting 2-3171
LU factorization 2-2234
gca 2-1493
gcbf function 2-1494
gcbo function 2-1495
gcd 2-1496
gcf 2-1498
gco 2-1499
ge 2-1500
generalized eigenvalue problem 2-1045 2-2833
generating a sequence of matrix names (M1 through M12) 2-1115
genpath 2-1502
genvarname 2-1504
geodesic dome 2-3645
get 2-1508 2-1511
memmapfile object 2-1514
serial port I/O 2-1518
timer object 2-1520
get (timeseries) 2-1522
get (tscollection) 2-1523
get hgsetget class method 2-1513
get, RandStream method 2-1517
getabstime (timeseries) 2-1524
getabstime (tscollection) 2-1526
getAllPackages meta.package method 2-2351
getappdata function 2-1528
getCompilerConfigurations 2-2369
getdatasamplesize 2-1531
getDefaultStream, RandStream method 2-1532
getdisp hgsetget class method 2-1533
getenv 2-1534
getfield 2-1535
getframe 2-1537
image resolution and 2-1538
getinterpmethod 2-1543
getpixelposition 2-1544
getpref function 2-1546
getqualitydesc 2-1548
getReport, MException method 2-1549
getsampleusingtime (timeseries) 2-1552
getsampleusingtime (tscollection) 2-1553
gettimeseriesnames 2-1554
gettsafteratevent 2-1555
gettsafterevent 2-1556
gettsatevent 2-1557
gettsbeforeatevent 2-1558
gettsbeforeevent 2-1559
gettsbetweenevents 2-1560
GIF files
writing 2-1818
ginput function 2-1565
global 2-1567
global variable
  defining 2-1567
global variables, clearing from workspace 2-614
gmres 2-1569
golden section search 2-1363
Goup
  defining default properties 2-1702
gplot 2-1575
grabcode function 2-1577
gradient 2-1579
gradient, numerical 2-1579
graph
  adjacency 2-1013
graph theory 2-4033
graphics objects
  Axes 2-290
  Figure 2-1233
  getting properties 2-1508
  Image 2-1760
  Light 2-2098
  Line 2-2111
  Patch 2-2702
  resetting properties 2-3111
Index

Root 2-3143
  setting properties 2-3238
Surface 2-3563
  Text 2-3670
  uicontextmenu 2-3831
  Uicontrol 2-3842
  Uimenu 2-3895
graphics objects, deleting 2-942
  graphs
    editing 2-2811
graymon 2-1582
  greatest common divisor 2-1496
Greek letters and mathematical symbols 2-196
    2-208 2-3699
grid 2-1583
    aligning data to a 2-1585
  grid arrays
    for volumetric plots 2-2333
    multi-dimensional 2-2492
griddata 2-1585
  griddatan 2-1589
  griddatan 2-1592
GridLineStyle
    hggroup property 2-1671
  hggroup function 2-1671
  gsvd 2-1595
gt 2-1601
gtext 2-1603
guidata function 2-1604
GUIDE
    object methods
      inspect 2-1866
guihandles function 2-1609
  GUIs, printing 2-2869
  gunzip 2-1610
gzip 2-1612

H
  H1 line 2-1656
hadamard 2-1613
  Hadamard matrix 2-1613
    subspaces of 2-3540
handle class 2-1614
handle graphics
  hgtransform 2-1691
  handle graphics hggroup 2-1671
  handle relational operators 2-3095
  handle.addlistener 2-118
  handle.delete 2-947
  handle.findobj 2-1337
  handle.findprop 2-1338
  handle.isValid 2-2013
  handle.notify 2-2575
HandleVisibility
  areasersies property 2-229
  Axes property 2-315
  barseries property 2-364
  contour property 2-721
  errorbar property 2-1096
  Figure property 2-1248
  hggroup property 2-1681
  hgtransform property 2-1710
  Image property 2-1778
  Light property 2-2103
  Line property 2-2125
  lineseries property 2-2138
  patch property 2-2730
  quivergroup property 2-2979
  rectangle property 2-3050
  Root property 2-3148
  stairseries property 2-3393
  stem property 2-3427
  Surface property 2-3582
  surfaceplot property 2-3605
  Text property 2-3688
  uicontextmenu property 2-3838
  Uicontrol property 2-3859
  Uimenu property 2-3903
  Uipushtool property 2-3938
Uitable property 2-3989
Uitoggletool property 2-4005
Uitoolbar property 2-4016

hankel 2-1618
Hankel matrix 2-1618

HDF
appending to when saving (WriteMode) 2-1821
compression 2-1821
setting JPEG quality when writing 2-1821

HDF files
writing images 2-1818

HDF4
summary of capabilities 2-1619

HDF5
high-level access 2-1621
summary of capabilities 2-1621

HDF5 class
low-level access 2-1621

hdf5info 2-1624
hdf5read 2-1626
hdf5write 2-1628
hdffinfo 2-1632
hdfread 2-1640
hdftool 2-1653

Head1Length
annotation doublearrow property 2-177

Head1Style
annotation doublearrow property 2-178

Head1Width
annotation doublearrow property 2-179

Head2Length
annotation doublearrow property 2-177

Head2Style
annotation doublearrow property 2-178

Head2Width
annotation doublearrow property 2-179

HeadLength
annotation arrow property 2-173
textarrow property 2-193

HeadStyle
annotation arrow property 2-174
textarrow property 2-194

HeadWidth
annotation arrow property 2-174
textarrow property 2-194

Height
annotation ellipse property 2-183

help 2-1654
contents file 2-1655
keyword search in functions 2-2215
online 2-1654

Help browser 2-1658
accessing from doc 2-1015

Help Window 2-1664
helpbrowser 2-1658
helpdesk 2-1661
helpdlg 2-1662
helpwin 2-1664

Hermite transformations, elementary 2-1496

Hess 2-1665
Hessenberg form of a matrix 2-1665

hex2dec 2-1668
hex2num 2-1669

hgsetget class 2-1690
hgsetget.get 2-1513
hgsetget.getdisp 2-1533
hgsetget.set 2-3243
hidden 2-1715

Hierarchical Data Format (HDF) files
writing images 2-1818

hilb 2-1716
Hilbert matrix 2-1716
inverse 2-1920

hist 2-1717
histc 2-1721

HitTest
areaseries property 2-231
Axes property 2-316
barseries property 2-366
Index-28

contour property 2-723
errorbar property 2-1098
Figure property 2-1250
hggroup property 2-1682
hgtransform property 2-1711
Image property 2-1780
Light property 2-2105
Line property 2-2125
lineseries property 2-2140
Patch property 2-2731
quivergroup property 2-2981
rectangle property 2-3051
Root property 2-3148
scatter property 2-3210
stairseries property 2-3395
stem property 2-3429
Surface property 2-3583
surfaceplot property 2-3607
Text property 2-3689
Uicontrol property 2-3860
Uipushtool property 2-3939
Uitable property 2-3989
Uitoggletool property 2-4005
Uitoolbarl property 2-4017

HitTestArea

areaseries property 2-231
barseries property 2-366
contour property 2-723
errorbar property 2-1098
quivergroup property 2-2981
scatter property 2-3210
stairseries property 2-3395
stem property 2-3429

hold 2-1724
home 2-1726

HorizontalAlignment

Text property 2-3690
textarrow property 2-194
textbox property 2-206
Uicontrol property 2-3861
horzcat 2-1727
horzcat (M-file function equivalent for [ , ] ) 2-65
horzcat (tscollection) 2-1729
hostid 2-1730
Householder reflections (as algorithm for solving linear equations) 2-2411
hsv2rgb 2-1731
HTML

in Command Window 2-2268
save M-file as 2-2903

HTML browser

in MATLAB 2-1658
hyperbolic

cosecant 2-789
cosecant, inverse 2-90
cosine 2-766
cosine, inverse 2-80
cotangent 2-771
cotangent, inverse 2-85
cscant 2-3226
cscant, inverse 2-248
cscant 2-3286
cscant, inverse 2-253
tangent 2-3659
tangent, inverse 2-264

hyperlink

displaying in Command Window 2-990
hyperlinks

in Command Window 2-2268
hyperplanes, angle between 2-3540
hypot 2-1732

I

i 2-1735
icon images

reading 2-1805
idealfilter (timeseries) 2-1736
identity matrix 2-1147

sparse 2-3336
idivide 2-1739
IEEE floating-point arithmetic
   smallest positive number 2-3032
if 2-1741
ifft 2-1745
ifft2 2-1747
ifftn 2-1749
ifftshift 2-1751
IIR filter 2-1319
ilu 2-1752
im2java 2-1757
imag 2-1759
image 2-1760
Image
   creating 2-1760
   properties 2-1768
image types
   querying 2-1791
images
   file formats 2-1803 2-1816
   reading data from files 2-1803
   returning information about 2-1790
   writing to files 2-1816
Images
   converting MATLAB image to Java
   Image 2-1757
imagesc 2-1784
imaginary 2-1759
   part of complex number 2-1759
   unit (\sqrt(-1)) 2-1735 2-2018
   See also complex
imapprox 2-1788
imfinfo
   returning file information 2-1790
imformats 2-1794
import 2-1797
importdata 2-1800
importing
   Java class and package names 2-1797
imread 2-1803
imwrite 2-1816
incomplete beta function
   (defined) 2-399
incomplete gamma function
   (defined) 2-1490
ind2sub 2-1838
Index into matrix is negative or zero (error message) 2-2207
indexed images
   converting from RGB 2-3127
indexing
   logical 2-2206
   indicator of file position 2-1420
indices, array
   of sorted elements 2-3311
Inf 2-1842
infeq 2-1844
infinity 2-1842
   norm 2-2570
info 2-1846
information
   returning file information 2-1790
inheritance, of objects 2-609
inline 2-1847
inmem 2-1850
inpolygon 2-1852
input 2-1854
   checking number of M-file arguments 2-2483
   name of array passed as 2-1859
   number of M-file arguments 2-2485
   prompting users for 2-1854
inputdlg 2-1855
inputname 2-1859
inputParser 2-1860
inspect 2-1866
installation, root directory of 2-2274
instance properties 2-129
instrcallback 2-1874
instrfind 2-1875
instrfindall 2-1877
example of 2-1878
int2str 2-1880
integer
  floating-point, maximum 2-429
IntegerHandle
  Figure property 2-1250
integration
  polynomial 2-2839
  quadrature 2-2937 2-2946
interfaces 2-1884
interp 2-1886
interp1 2-1894
interp2 2-1896
interp3 2-1900
interpft 2-1902
interpn 2-1903
interpolated shading and printing 2-2871
interpolation
  cubic method 2-1585 2-1886 2-1896 2-1900
  2-1903
  cubic spline method 2-1886 2-1896 2-1900
  2-1903
  FFT method 2-1902
linear method 2-1886 2-1896 2-1900 2-1903
multidimensional 2-1903
nearest neighbor method 2-1585 2-1886
  2-1896 2-1900 2-1903
one-dimensional 2-1886
three-dimensional 2-1900
trilinear method 2-1585
two-dimensional 2-1896
Interpreter
  Text property 2-3691
textarrow property 2-194
textbox property 2-207
interpstreamspeed 2-1906
Interruptible
  areaseries property 2-231
  Axes property 2-316
  barseries property 2-366
  contour property 2-723
  errorbar property 2-1099
  Figure property 2-1250
  hggroup property 2-1682
  hgtransform property 2-1711
  Image property 2-1780
  Light property 2-2105
  Line property 2-2126
  lineseries property 2-2140
  patch property 2-2731
  quivergroup property 2-2981
  rectangle property 2-3052
  Root property 2-3148
  scatter property 2-3211
  stairseries property 2-3395
  stem property 2-3430
  Surface property 2-3583 2-3607
  Text property 2-3692
  Uicontextmenu property 2-3839
  Uicontrol property 2-3861
  Uimenu property 2-3904
  Uipushtool property 2-3939
  Uitable property 2-3990
  Uitoggletool property 2-4006
  Uitoolbar property 2-4017
intersect 2-1910
intmax 2-1911
intmin 2-1912
intwarning 2-1913
inv 2-1917
inverse
  cosecant 2-87
  cosine 2-77
  cotangent 2-82
  Fourier transform 2-1745 2-1747 2-1749
  Hilbert matrix 2-1920
  hyperbolic cosecant 2-90
  hyperbolic cosine 2-80
  hyperbolic cotangent 2-85
  hyperbolic secant 2-248
hyperbolic sine 2-253
hyperbolic tangent 2-264
of a matrix 2-1917
secant 2-245
sine 2-250
tangent 2-259
tangent, four-quadrant 2-261
inversion, matrix
accuracy of 2-688
InvertHardCopy, Figure property 2-1251
invhilb 2-1920
invoke 2-1921
involuntary matrix 2-2701
ipermute 2-1924
iqr (timeseries) 2-1925
is* 2-1927
isa 2-1930
isappdata function 2-1932
iscell 2-1933
iscellstr 2-1934
ischar 2-1935
iscom 2-1936
isdir 2-1937
isempty 2-1940
isempty (timeseries) 2-1941
isempty (tscollection) 2-1942
isequal 2-1943
isequal, MException method 2-1946
isequalwithequalitynans 2-1947
isevent 2-1949
isfield 2-1951
isfinite 2-1953
isfloat 2-1954
isglobal 2-1955
ishandle 2-1957
isin 2-1959
isinteger 2-1960
isinterface 2-1961
isjava 2-1962
iskeyword 2-1965
isletter 2-1967
islogical 2-1968
ismac 2-1969
ismember 2-1970
ismethod 2-1972
isnan 2-1973
isnumeric 2-1974
isobject 2-1975
isocap 2-1977
isonormals 2-1984
isosurface 2-1987
calculate data from volume 2-1987
end caps 2-1977
vertex normals 2-1984
ispc 2-1992
isPlatformSupported 2-2436
ispref function 2-1993
isprime 2-1994
isprop 2-1995
isreal 2-1996
isscalar 2-1999
issorted 2-2000
isspace 2-2003 2-2006
issparse 2-2004
isstr 2-2005
isstruct 2-2009
isstudent 2-2010
isunix 2-2011 to 2-2012
isvalid 2-2014
timer object 2-2015
isvalid handle method 2-2013
isvarname 2-2016
isvector 2-2017
italics font
TeX characters 2-3701

J
j 2-2018
Jacobi rotations 2-3358
Jacobian elliptic functions
(defined) 2-1060
Jacobian matrix (BVP) 2-479
Jacobian matrix (ODE) 2-2628
  generating sparse numerically 2-2629
  2-2631
  specifying 2-2628 2-2631
  vectorizing ODE function 2-2629 to 2-2631
Java
  class names 2-616 2-1797
  object methods
    inspect 2-1866
    objects 2-1962
Java Image class
  creating instance of 2-1757
Java import list
  adding to 2-1797
  clearing 2-616
Java version used by MATLAB 2-4102
java_method 2-2023 2-2031
java_object 2-2034
javaaddath 2-2019
javachk 2-2024
javaclasspath 2-2026
javaMethod 2-2031
javaMethodEDT 2-2033
javaObject 2-2034
javaObjectEDT 2-2036
javarmpath 2-2037
joining arrays. See concatenation
Joint Photographic Experts Group (JPEG)
  writing 2-1818
JPEG
  setting Bitdepth 2-1822
  specifying mode 2-1822
JPEG comment
  setting when writing a JPEG image 2-1822
JPEG files
  parameters that can be set when writing 2-1822
writing 2-1818
JPEG quality
  setting when writing a JPEG image 2-1822
  2-1826
  setting when writing an HDF image 2-1821
jvm
  version used by MATLAB 2-4102

K
K>> prompt
  keyboard function 2-2041
keep
  some variables when clearing 2-620
keyboard 2-2041
keyboard mode 2-2041
  terminating 2-3125
KeyPressFcn
  Uicontrol property 2-3862
  Uitable property 2-3991
KeyPressFcn, Figure property 2-1252
KeyReleaseFcn, Figure property 2-1253
keyword search in functions 2-2215
keywords
  iskeyword function 2-1965
kron 2-2043
Kronecker tensor product 2-2043
Krylov subspaces 2-3727

L
Label, Uimenu property 2-3905
labeling
  axes 2-4200
  matrix columns 2-990
  plots (with numeric values) 2-2584
LabelSpacing
  contour property 2-724
Laplacian 2-917
Laplacian matrix 2-4033
largest array elements 2-2299
last, MException method 2-2045
lasterr 2-2047
lasterror 2-2050
lastwarn 2-2055
LaTeX, see TeX 2-196 2-208 2-3699
Layer, Axes property 2-317
Layout Editor
  starting 2-1608
1cm 2-2057
LData
  errorbar property 2-1099
LDataSource
  errorbar property 2-1099
ldivide (M-file function equivalent for .\) 2-45
le 2-2065
least common multiple 2-2057
least squares
  polynomial curve fitting 2-2835
  problem, overdetermined 2-2794
legend 2-2067
  properties 2-2072
  setting text properties 2-2072
legendre 2-2076
Legendre functions
  (defined) 2-2076
  Schmidt semi-normalized 2-2076
length 2-2080
  serial port I/O 2-2082
length (timeseries) 2-2083
length (tscollection) 2-2084
LevelList
  contour property 2-724
LevelListMode
  contour property 2-724
LevelStep
  contour property 2-725
LevelStepMode
  contour property 2-725
libfunctions 2-2085
libfunctionsview 2-2086
libisloaded 2-2087
libpointer 2-2089
libstruct 2-2091
license 2-2094
light 2-2098
Light
  creating 2-2098
  defining default properties 2-1766 2-2099
  positioning in camera coordinates 2-495
  properties 2-2100
Light object
  positioning in spherical coordinates 2-2108
lightangle 2-2108
lighting 2-2109
limits of axes, setting and querying 2-4202
line 2-2111
  editing 2-2811
Line
  creating 2-2111
  defining default properties 2-2116
  properties 2-2117 2-2132
line numbers in M-files 2-881
linear audio signal 2-2110 2-2466
linear dependence (of data) 2-3540
linear equation systems
  accuracy of solution 2-688
  solving overdetermined 2-2927 to 2-2928
linear equation systems, methods for solving
  Cholesky factorization 2-2409
  Gaussian elimination 2-2410
  Householder reflections 2-2411
  matrix inversion (inaccuracy of) 2-1917
linear interpolation 2-1886 2-1896 2-1900 2-1903
linear regression 2-2835
linearly spaced vectors, creating 2-2174
LineColor
  contour property 2-725
lines
  computing 2-D stream 2-3457
computing 3-D stream 2-3459

drawing stream lines 2-3461

LineSpec 2-2149

LineStyle

  annotation arrow property 2-174
  annotation doublearrow property 2-179
  annotation ellipse property 2-183
  annotation line property 2-185
  annotation rectangle property 2-189
  annotation textbox property 2-207
  area series property 2-232
  bar series property 2-367
  contour property 2-726
  errorbar property 2-1100
  Line property 2-2127
  lineseries property 2-2141
  patch property 2-2732
  quiver group property 2-2982
  rectangle property 2-3052
  stair series property 2-3396
  stem property 2-3431
  Surface property 2-3584
  surface plot property 2-3608
  text object 2-3694
  text arrow property 2-195

LineStyleOrder

  Axes property 2-317

LineWidth

  annotation arrow property 2-175
  annotation doublearrow property 2-180
  annotation ellipse property 2-183
  annotation line property 2-186
  annotation rectangle property 2-189
  annotation textbox property 2-207
  area series property 2-232
  Axes property 2-318
  bar series property 2-367
  contour property 2-726
  errorbar property 2-1100
  Line property 2-2127
  lineseries property 2-2141
  Patch property 2-2732
  quiver group property 2-2982
  rectangle property 2-3052
  scatter property 2-3211
  stair series property 2-3396
  stem property 2-3431
  Surface property 2-3584
  surface plot property 2-3608
  text object 2-3694
  text arrow property 2-195

linkaxes 2-2155
linkdata 2-2159
linkprop 2-2167
links
  in Command Window 2-2268
linsolve 2-2171
linspace 2-2174
lint tool for checking problems 2-2413
list boxes 2-3844
  defining items 2-3868
list, RandStream method 2-2175
ListboxTop, Uicontrol property 2-3863
listdlg 2-2178
listfonts 2-2181
little endian formats 2-1373
load 2-2183 2-2188
  serial port I/O 2-2190
loadlibrary 2-2192
loadobj 2-2200
Lobatto IIIa ODE solver 2-465 2-471
local variables 2-1441 2-1567
locking M-files 2-2427
log 2-2202
  saving session to file 2-977
log10 [log010] 2-2203
log1p 2-2204
log2 2-2205

  logarithm
  base ten 2-2203
base two 2-2205
complex 2-2202 to 2-2203
natural 2-2202
of beta function (natural) 2-402
of gamma function (natural) 2-1491
of real numbers 2-3030
plotting 2-2208
logarithmic derivative
gamma function 2-2900
logarithmically spaced vectors, creating 2-2214
logical 2-2206
logical array
converting numeric array to 2-2206
detecting 2-1968
logical indexing 2-2206
logical operations
AND, bit-wise 2-424
OR, bit-wise 2-431
XOR 2-4228
XOR, bit-wise 2-435
logical operators 2-52 2-59
logical OR
bit-wise 2-431
logical tests 2-1930
all 2-153
any 2-213
See also detecting
logical XOR 2-4228
bit-wise 2-435
loglog 2-2208
logm 2-2211
logspace 2-2214
lookfor 2-2215
lossy compression
writing JPEG files with 2-1822
Lotus WK1 files
loading 2-4193
writing 2-4195
lower 2-2217
lower triangular matrix 2-3766
lowercase to uppercase 2-4054
ls 2-2218
lscov 2-2219
lsqnonneg 2-2224
lsqr 2-2227
lt 2-2232
lu 2-2234
LU factorization 2-2234
storage requirements of (sparse) 2-2588
luinc 2-2242
M
M-file
debugging 2-2041
displaying during execution 2-1034
function 2-1441
function file, echoing 2-1034
naming conventions 2-1441
pausing execution of 2-2745
programming 2-1441
script 2-1441
script file, echoing 2-1034
M-file execution
resuming after suspending 2-3953
suspending from GUI 2-4020
M-files
checking existence of 2-1135
checking for problems 2-2413
clearing from workspace 2-614
cyclomatic complexity of 2-2413
debugging with profile 2-2887
deleting 2-942
editing 2-1041
line numbers, listing 2-881
lint tool 2-2413
listing names of in a directory 2-4167
locking (preventing clearing) 2-2427
McCabe complexity of 2-2413
opening 2-2642
optimizing 2-2887
problems, checking for 2-2413
save to HTML 2-2903
setting breakpoints 2-870
unlocking (allowing clearing) 2-2478
M-Lint
  function 2-2413
  function for entire directory 2-2423
  HTML report 2-2423
machine epsilon 2-4177
magic 2-2249
magic squares 2-2249
Map containers
  constructor 2-2254 2-3291
  methods 2-2081 2-3098 2-4085
Map methods
  constructor 2-1963 2-2042
Margin
  annotation textbox property 2-208
  text object 2-3696
Marker
  Line property 2-2127
  lineseries property 2-2141
  marker property 2-1101
  Patch property 2-2732
  quivergroup property 2-2982
  scatter property 2-3212
  stairseries property 2-3396
  stem property 2-3431
  Surface property 2-3584
  surfaceplot property 2-3608
MarkerEdgeColor
  errorbar property 2-1101
  Line property 2-2128
  lineseries property 2-2142
  Patch property 2-2733
  quivergroup property 2-2983
  scatter property 2-3212
  stairseries property 2-3397
  stem property 2-3432
  Surface property 2-3585
  surfaceplot property 2-3609
MarkerFaceColor
  errorbar property 2-1102
  Line property 2-2128
  lineseries property 2-2142
  Patch property 2-2734
  quivergroup property 2-2983
  scatter property 2-3213
  stairseries property 2-3397
  stem property 2-3432
  Surface property 2-3586
  surfaceplot property 2-3609
MarkerSize
  errorbar property 2-1102
  Line property 2-2129
  lineseries property 2-2143
  Patch property 2-2734
  quivergroup property 2-2984
  stairseries property 2-3398
  stem property 2-3432
  Surface property 2-3586
  surfaceplot property 2-3610
mass matrix (ODE) 2-2632
  initial slope 2-2633 to 2-2634
  singular 2-2633
  sparsity pattern 2-2633
  specifying 2-2633
  state dependence 2-2633
MAT-file 2-3176
  converting sparse matrix after loading from 2-3323
MAT-files 2-2183
  listing for directory 2-4167
mat2cell 2-2261
mat2str 2-2264
material 2-2266
MATLAB
  directory location 2-2274
  installation directory 2-2274
quitting 2-2962
startup 2-2273
version number, comparing 2-4100
version number, displaying 2-4094
matlab : function 2-2268
matlab (UNIX command) 2-2277
matlab (Windows command) 2-2292
matlab function for UNIX 2-2277
matlab function for Windows 2-2292
MATLAB startup file 2-3406
MATLAB® desktop
  moving figure windows in front of 2-3273
matlab.mat 2-2183 2-3176
matlabcolon function 2-2268
matlabrc 2-2273
matlabroot 2-2274
$matlabroot 2-2274
matrices
  preallocation 2-4231
matrix 2-40
  addressing selected rows and columns
  of 2-66
  arrowhead 2-673
  columns
    rearrange 2-1355
  companion 2-681
  complex unitary 2-2925
  condition number of 2-688 2-3023
  condition number, improving 2-340
  converting to formatted data file 2-1391
  converting to from string 2-3376
  converting to vector 2-66
  decomposition 2-2925
  defective (defined) 2-1046
  detecting sparse 2-2004
  determinant of 2-967
  diagonal of 2-973
  Dulmage-Mendelsohn decomposition 2-1012
  evaluating functions of 2-1450
  exponential 2-1142
Hadamard 2-1613 2-3540
Hankel 2-1618
Hermitian Toeplitz 2-3756
Hessenberg form of 2-1665
Hilbert 2-1716
identity 2-1147
inverse 2-1917
inverse Hilbert 2-1920
inversion, accuracy of 2-688
involuntary 2-2701
left division (arithmetic operator) 2-41
lower triangular 2-3766
magic squares 2-2249 2-3548
maximum size of 2-686
modal 2-1044
multiplication (defined) 2-41
orthonormal 2-2925
Pascal 2-2701 2-2842
permutation 2-2234 2-2925
poorly conditioned 2-1716
power (arithmetic operator) 2-42
pseudoinverse 2-2794
reading files into 2-1004
rearrange
  columns 2-1355
  rows 2-1356
reduced row echelon form of 2-3171
replicating 2-3103
right division (arithmetic operator) 2-41
rotating 90\xfb 2-3160
rows
  rearrange 2-1356
Schur form of 2-3173 2-3219
singularity, test for 2-967
sorting rows of 2-3314
sparse. See sparse matrix
specialized 2-1467
square root of 2-3370
subspaces of 2-3540
test 2-1467
Index

Toeplitz 2-3756
trace of 2-973 2-3758
transpose (arithmetic operator) 2-42
transposing 2-63
unimodular 2-1496
unitary 2-3624
upper triangular 2-3785
Vandermonde 2-2837
Wilkinson 2-3329 2-4188
writing as binary data 2-1455
writing formatted data to 2-1421
writing to ASCII delimited file 2-1008
writing to spreadsheet 2-4195
See also array

Matrix
hgtransform property 2-1712
matrix functions
evaluating 2-1450
matrix names, (M1 through M12) generating a
sequence of 2-1115
matrix power. See matrix, exponential
max 2-2299
max (timeseries) 2-2300
Max, Uicontrol property 2-3863
MaxHeadSize
quivergroup property 2-2984
maximum matching 2-1012
MDL-files
checking existence of 2-1135
mean 2-2305
mean (timeseries) 2-2306
median 2-2308
median (timeseries) 2-2309
median value of array elements 2-2308
memmap file 2-2311
memory 2-2317
   clearing 2-614
   minimizing use of 2-2678
   variables in 2-4181
menu (of user input choices) 2-2326
menu function 2-2326
MenuBar, Figure property 2-1256
Mersenne twister 2-3010 2-3014
mesh plot
tetrahedron 2-3665
mesh size (BVP) 2-482
meshc 2-2328
meshgrid 2-2333
MeshStyle, Surface property 2-3586
MeshStyle, surfaceplot property 2-3610
meshz 2-2328
message
error See error message 2-4147
warning See warning message 2-4147
meta.class 2-2335
meta.DynamicProperty 2-2340
meta.event 2-2344
meta.method 2-2346
meta.package class 2-2349
meta.property 2-2352
methods 2-2357
   inheritance of 2-609
   locating 2-4172
methodsview 2-2359
mex 2-2361
mex build script
   switches 2-2362
      -ada sfcn.ads 2-2363
      -arch 2-2363
      -argcheck 2-2363
      -c 2-2363
      -compatibleArrayDims 2-2363
      -cxx 2-2364
      -Dname 2-2364
      -Dname=value 2-2364
      -f optionsfile 2-2364
      -fortran 2-2364
      -g 2-2364
      -h[elp] 2-2364
      -inline 2-2365
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Ipathname</td>
<td>2-2365</td>
</tr>
<tr>
<td>-largeArrayDims</td>
<td>2-2365</td>
</tr>
<tr>
<td>-Ldirectory</td>
<td>2-2365</td>
</tr>
<tr>
<td>-lname</td>
<td>2-2365</td>
</tr>
<tr>
<td>-n</td>
<td>2-2365</td>
</tr>
<tr>
<td>name=value</td>
<td>2-2366</td>
</tr>
<tr>
<td>-o</td>
<td>2-2366</td>
</tr>
<tr>
<td>-outdir dirname</td>
<td>2-2366</td>
</tr>
<tr>
<td>-output resultname</td>
<td>2-2366</td>
</tr>
<tr>
<td>@rsp_file</td>
<td>2-2362</td>
</tr>
<tr>
<td>-setup</td>
<td>2-2366</td>
</tr>
<tr>
<td>-Uname</td>
<td>2-2366</td>
</tr>
<tr>
<td>-v</td>
<td>2-2366</td>
</tr>
</tbody>
</table>

mex.CompilerConfiguration 2-2369
mex.CompilerConfigurationDetails 2-2369
MEX-files
   clearing from workspace 2-614
   debugging on UNIX 2-860
   listing for directory 2-4167
mex.getCompilerConfigurations 2-2369
MException
   constructor 2-1078 2-2375
   methods
      addCause 2-110
      disp 2-993
      eq 2-1078
      getReport 2-1549
      isequal 2-1946
      last 2-2045
      ne 2-2499
      rethrow 2-3123
      throw 2-3730
      throwAsCaller 2-3733
mexext 2-2382
mfilename 2-2383
mget function 2-2384
Microsoft Excel files
   loading 2-4207
min 2-2385
min (timeseries) 2-2386

Min, Uicontrol property 2-3864
MinColormap, Figure property 2-1256
MinorGridLineStyle, Axes property 2-319
minres 2-2390
minus (M-file function equivalent for -) 2-45
mislocked 2-2395
mkdir 2-2396
mkdir (ftp) 2-2399
mkpp 2-2400
mldivide (M-file function equivalent for \) 2-45
mlint 2-2413
mlintrpt 2-2423
suppressing messages 2-2426
mlock 2-2427
mmfileinfo 2-2428
mmreader 2-2431
mmreader.isPlatformSupported 2-2436
mod 2-2437
modal matrix 2-1044
mode 2-2439
mode objects
   pan, using 2-2683
   rotate3d, using 2-3164
   zoom, using 2-4237
models
   opening 2-2642
   saving 2-3188
modification date
   of a file 2-982
modified Bessel functions
   relationship to Airy functions 2-143
modulo arithmetic 2-2437
MonitorPosition
   Root property 2-3148
Moore-Penrose pseudoinverse 2-2794
more 2-2442 2-2466
move 2-2444
movefile 2-2446
movegui function 2-2449
movie 2-2451
movie2avi 2-2455
movies
   exporting in AVI format 2-283
mpower (M-file function equivalent for ^) 2-46
mput function 2-2458
mrdivide (M-file function equivalent for '/') 2-45
msgbox 2-2459
mtimes 2-2462
mtimes (M-file function equivalent for *) 2-45
mu-law encoded audio signals 2-2110 2-2466
multibandread 2-2467
multibandwrite 2-2472
multidimensional arrays 2-2080
   concatenating 2-522
   interpolation of 2-1903
   longest dimension of 2-2080
   number of dimensions of 2-2494
   rearranging dimensions of 2-1924 2-2785
   removing singleton dimensions of 2-3373
   reshaping 2-3114
   size of 2-3288
   sorting elements of 2-3310
See also array
multiple
   least common 2-2057
multiplication
   array (arithmetic operator) 2-41
   matrix (defined) 2-41
   of polynomials 2-740
multistep ODE solver 2-2609
munlock 2-2478

returned by rem 2-3097
nargchk 2-2483
nargoutchk 2-2487
native2unicode 2-2489
ndgrid 2-2492
ndims 2-2494
ne 2-2495
ne, MException method 2-2499
nearest neighbor interpolation 2-1585 2-1886
   2-1896 2-1900 2-1903
.NET
   summary of functions 2-2502
.NET
   summary of functions 2-2502
netcdf
   summary of capabilities 2-2513 2-2545
netcdf.abort
   revert recent netCDF file definitions 2-2516
netcdf.close
   close netCDF file 2-2518
netcdf.copyAtt
   copy attribute to new location 2-2519
netcdf.create
   create netCDF file 2-2521
netcdf.defDim
   create dimension in netCDF file 2-2523
netcdf.defVar
   define variable in netCDF dataset 2-2524
netcdf.delAtt
   delete netCDF attribute 2-2525
netcdf.endDef
   takes a netCDF file out of define mode 2-2527
netcdf.getAtt
   return data from netCDF attribute 2-2529
netcdf.getConstant
   get numeric value of netCDF constant 2-2531
netcdf.getConstantNames
   get list of netCDF constants 2-2532
netcdf.getVar
   return data from netCDF variable 2-2533

N
Name, Figure property 2-1257
namelengthmax 2-2480
naming conventions
   M-file 2-1441
NaN 2-2481
NaN (Not-a-Number) 2-2481

Index-40
netcdf.inq
  return information about netCDF file 2-2535
netcdf.inqAtt
  return information about a netCDF attribute 2-2537
netcdf.inqAttID
  return identifier of netCDF attribute 2-2539
netcdf.inqAttName
  return name of netCDF attribute 2-2540
netcdf.inqDim
  return information about netCDF dimension 2-2542
netcdf.inqDimID
  return dimension ID for netCDF file 2-2543
netcdf.inqLibVers
  return version of netCDF library 2-2544
netcdf.inqVarID
  return netCDF variable identifier 2-2547
netcdf.open
  open an existing netCDF file 2-2548
netcdf.putAtt
  write a netCDF attribute 2-2549
netcdf.putVar
  write data to netCDF variable 2-2551
netcdf.reDef
  put netCDF file into define mode 2-2553
netcdf.renameAtt
  netCDF function to change the name of an attribute 2-2554
netcdf.renameDim
  netCDF function to change the name of a dimension 2-2556
netcdf.renameVar
  change the name of a netCDF variable 2-2558
netcdf.setDefaultFormat
  change the default netCDF file format 2-2560
netcdf.setFill
  set netCDF fill behavior 2-2561
netcdf.sync
  synchronize netCDF dataset to disk 2-2562
newplot 2-2563
NextPlot
  Axes property 2-319
  Figure property 2-1257
nextpow2 2-2566
nnz 2-2567
no derivative method 2-1369
nodesktop startup option 2-2281
noncontiguous fields, inserting data into 2-1455
nonzero entries
  specifying maximum number of in sparse matrix 2-3320
nonzero entries (in sparse matrix)
  allocated storage for 2-2588
  number of 2-2567
  replacing with ones 2-3350
  vector of 2-2569
nonzeros 2-2569
norm 2-2570
  1-norm 2-2570 2-3023
  2-norm (estimate of) 2-2572
  F-norm 2-2570
  infinity 2-2570
  matrix 2-2570
  pseudoinverse and 2-2794 2-2796
  vector 2-2570
normal vectors, computing for volumes 2-1984
NormalMode
  Patch property 2-2734
  Surface property 2-3587
  surfaceplot property 2-3610
normest 2-2572
not 2-2573
not (M-file function equivalent for ~) 2-56
notebook 2-2574
notify 2-2575
now 2-2576
nthroot 2-2577
null 2-2578
null space 2-2578
num2cell 2-2580
num2hex 2-2583
num2str 2-2584
number
  of array dimensions 2-2494
numbers
  imaginary 2-1759
  NaN 2-2481
  plus infinity 2-1842
  prime 2-2853
  real 2-3029
  smallest positive 2-3032
NumberTitle, Figure property 2-1258
numel 2-2586
numeric format 2-1379
numeric precision
  format reading binary data 2-1403
numerical differentiation formula ODE
  solvers 2-2609
numerical evaluation
  double integral 2-858
  triple integral 2-3768
nzmax 2-2588

object
  determining class of 2-1930
  inheritance 2-609
object classes, list of predefined 2-608 2-1930
objects
  Java 2-1962
ODE file template 2-2612
ODE solver properties
  error tolerance 2-2619
  event location 2-2626
  Jacobian matrix 2-2628
  mass matrix 2-2632
  ode15s 2-2634
  solver output 2-2621
  step size 2-2625
ODE solvers
  backward differentiation formulas 2-2634
  numerical differentiation formulas 2-2634
  obtaining solutions at specific times 2-2596
  variable order solver 2-2634
ode15i function 2-2589
odefile 2-2611
odeget 2-2617
odephas2 output function 2-2623
odephas3 output function 2-2623
odeplot output function 2-2623
odeprint output function 2-2623
odeset 2-2618
odextend 2-2636
off-screen figures, displaying 2-1333
OffCallback
  Uitoggletool property 2-4007
%#ok 2-2416
OnCallback
  Uitoggletool property 2-4007
online documentation, displaying 2-1658
online help 2-1654
open 2-2642
openfig 2-2646
OpenGL 2-1265
  autoselection criteria 2-1269
opening
  files in Windows applications 2-4189
opening files 2-1370
openvar 2-2653
operating system
  MATLAB is running on 2-686
  operating system command 2-3653
  operating system command, issuing 2-65
operators
  arithmetic 2-40
logical 2-52 2-59
overloading arithmetic 2-46
overloading relational 2-50
relational 2-50 2-2206
symbols 2-1654
optimget 2-2657
optimization parameters structure 2-2657 to 2-2658
optimizing M-file execution 2-2887
optimset 2-2658
or 2-2662
or (M-file function equivalent for |) 2-56
ordeig 2-2664
orderfields 2-2667
ordering
    reverse Cuthill-McKee 2-3635 2-3645
ordqz 2-2670
ordschur 2-2672
orient 2-2674
orth 2-2676
orthogonal-triangular decomposition 2-2925
orthographic projection, setting and querying 2-504
orthonormal matrix 2-2925
otherwise 2-2677
Out of memory (error message) 2-2678
OuterPosition
    Axes property 2-319
    Figure property 2-1258
output
    checking number of M-file arguments 2-2487
    controlling display format 2-1379
    in Command Window 2-2442
    number of M-file arguments 2-2485
output points (ODE)
    increasing number of 2-2621
output properties (DDE) 2-897
output properties (ODE) 2-2621
    increasing number of output points 2-2621
overdetermined equation systems, solving 2-2927 to 2-2928
overflow 2-1842
overloading
    arithmetic operators 2-46
    relational operators 2-50
    special characters 2-65

P
P-files
    checking existence of 2-1135
pack 2-2678
padecoef 2-2680
pagesetupdlg 2-2681
paging
    of screen 2-1656
paging in the Command Window 2-2442
pan mode objects 2-2683
PaperOrientation, Figure property 2-1259
PaperPosition, Figure property 2-1259
PaperPositionMode, Figure property 2-1260
PaperSize, Figure property 2-1260
PaperType, Figure property 2-1260
PaperUnits, Figure property 2-1261
parametric curve, plotting 2-1168
Parent
    areaseries property 2-233
    Axes property 2-321
    barseries property 2-368
    contour property 2-726
    errorbar property 2-1102
    Figure property 2-1262
    hggroup property 2-1683
    hgtransform property 2-1712
    Image property 2-1780
    Light property 2-2105
    Line property 2-2129
    lineseries property 2-2143
    Patch property 2-2734
quivergroup property 2-2984
rectangle property 2-3052
Root property 2-3149
scatter property 2-3213
stairseries property 2-3398
stem property 2-3432
Surface property 2-3587
surfaceplot property 2-3611
Text property 2-3697
Uicontextmenu property 2-3840
Uicontrol property 2-3865
Uimenu property 2-3906
Uipushtool property 2-3941
Uitable property 2-3992
Uitoggletool property 2-4008
Uitoolbar property 2-4018
parentheses (special characters) 2-63
parfor 2-2693
parse method
  of inputParser object 2-2695
parseSoapResponse 2-2697
partial fraction expansion 2-3116
partialpath 2-2699
pascal 2-2701
Pascal matrix 2-2701 2-2842
patch 2-2702
Patch
  converting a surface to 2-3561
  creating 2-2702
  defining default properties 2-2712
  properties 2-2713
  reducing number of faces 2-3058
  reducing size of face 2-3277
path 2-2739
  adding directories to 2-126
  building from parts 2-1438
  current 2-2739
  removing directories from 2-3141
  toolbox directory 2-3757
  viewing 2-2743
path2rc 2-2741
pathname
  partial 2-2699
pathnames
  of functions or files 2-4172
  relative 2-2699
pathsep 2-2742
pathtool 2-2743
pause 2-2745
pauses, removing 2-853
pausing M-file execution 2-2745
pbaspect 2-2747
PBM
  parameters that can be set when
    writing 2-1822
PBM files
  writing 2-1818
pcg 2-2753
pchips 2-2757
pcode 2-2760
pcolor 2-2762
PCX files
  writing 2-1818
PDE. See Partial Differential Equations
pdepe 2-2766
pdeval 2-2778
percent sign (special characters) 2-64
percent-brace (special characters) 2-64
perfect matching 2-1012
performance 2-378
period (.), to distinguish matrix and array
  operations 2-40
period (special characters) 2-63
perl 2-2781
perl function 2-2781
Perl scripts in MATLAB 2-2781
perms 2-2784
permutation
  matrix 2-2234 2-2925
  of array dimensions 2-2785
Index

random 2-3008
permutations of n elements 2-2784
permute 2-2785
persistent 2-2786
persistent variable 2-2786
perspective projection, setting and querying 2-504
PGM
parameters that can be set when writing 2-1822
PGM files
writing 2-1819
phase angle, complex 2-168
phase, complex
   correcting angles 2-4047
pi 2-2789
pie 2-2790
pie3 2-2792
pinv 2-2794
planerot 2-2797
platform MATLAB is running on 2-686
playshow function 2-2798
plot 2-2799
   editing 2-2811
plot (timeseries) 2-2806
plot box aspect ratio of axes 2-2747
plot editing mode
   overview 2-2812
Plot Editor
   interface 2-2812 2-2895
plot, volumetric
   generating grid arrays for 2-2333
   slice plot 2-3297
PlotBoxAspectRatio, Axes property 2-321
PlotBoxAspectRatioMode, Axes property 2-321
plotedit 2-2811
plotting
   2-D plot 2-2799
   3-D plot 2-2807
   contours (a 2-1148
   contours (ez function) 2-1148
ez-function mesh plot 2-1156
feather plots 2-1191
filled contours 2-1152
function plots 2-1386
functions with discontinuities 2-1176
histogram plots 2-1717
in polar coordinates 2-1171
isosurfaces 2-1987
loglog plot 2-2208
mathematical function 2-1164
mesh contour plot 2-1160
mesh plot 2-2328
parametric curve 2-1168
plot with two y-axes 2-2818
ribbon plot 2-3132
rose plot 2-3156
scatter plot 2-2814
scatter plot, 3-D 2-3198
semilogarithmic plot 2-3229
stem plot, 3-D 2-3417
surface plot 2-3555
surfaces 2-1174
velocity vectors 2-692
volumetric slice plot 2-3297
. See visualizing
plus (M-file function equivalent for +) 2-45
PNG
writing options for 2-1823
   alpha 2-1823
   background color 2-1823
   chromaticities 2-1824
   gamma 2-1824
   interlace type 2-1824
   resolution 2-1825
   significant bits 2-1824
   transparency 2-1825
PNG files
writing 2-1819
PNM files
writing 2-1819
Pointer, Figure property 2-1262
PointerLocation, Root property 2-3149
PointerShapeCData, Figure property 2-1262
PointerShapeHotSpot, Figure property 2-1263
PointerWindow, Root property 2-3150
pol2cart 2-2823
polar 2-2825
polar coordinates 2-2823
computing the angle 2-168
converting from Cartesian 2-516
converting to cylindrical or Cartesian 2-2823
plotting in 2-1171
poles of transfer function 2-3116
poly 2-2827
polyarea 2-2830
polyder 2-2832
polyeig 2-2833
polyfit 2-2835
polygamma function 2-2900
polygon
area of 2-2830
creating with patch 2-2702
detecting points inside 2-1852
polyint 2-2839
polynomial
analytic integration 2-2839
characteristic 2-2827 to 2-2828 2-3154
coefficients (transfer function) 2-3116
curve fitting with 2-2835
derivative of 2-2832
division 2-916
eigenvalue problem 2-2833
evaluation 2-2840
evaluation (matrix sense) 2-2842
make piecewise 2-2400
multiplication 2-740
polyval 2-2840
polyvalm 2-2842
poorly conditioned
matrix 2-1716
poorly conditioned eigenvalues 2-340
pop-up menus 2-3844
pop-up menus, defining choices 2-3868
Portable Anymap files
writing 2-1819
Portable Bitmap (PBM) files
writing 2-1818
Portable Graymap files
writing 2-1819
Portable Network Graphics files
writing 2-1819
Portable pixmap format
writing 2-1819
Position
annotation ellipse property 2-183
annotation line property 2-186
annotation rectangle property 2-190
arrow property 2-175
Axes property 2-322
doublearrow property 2-180
Figure property 2-1263
Light property 2-2105
Text property 2-3697
textarrow property 2-195
textbox property 2-208
Uicontextmenu property 2-3840
Uicontrol property 2-3865
Uimenu property 2-3906
Uitable property 2-3992
position indicator in file 2-1310 2-1434
position of camera
dollying 2-491
position of camera, setting and querying 2-502
Position, rectangle property 2-3053
PostScript
default printer 2-2860
levels 1 and 2 2-2860
printing interpolated shading 2-2871
pow2 2-2844
power 2-2845
  matrix. See matrix exponential
  of real numbers 2-3033
  of two, next 2-2566
power (M-file function equivalent for .^) 2-46
PPM
  parameters that can be set when
  writing 2-1822
PPM files
  writing 2-1819
ppval 2-2846
preallocation
  matrix 2-4231
precision 2-1379
  reading binary data writing 2-1403
prefdir 2-2848
preferences 2-2852
  opening the dialog box 2-2852
prime factors 2-1185
  dependence of Fourier transform on 2-1209
  2-1211 to 2-1212
prime numbers 2-2853
primes 2-2853
printdlg 2-2875
printdlg function 2-2875
printer
  default for linux and unix 2-2860
printer drivers
  GhostScript drivers 2-2856
  interpolated shading 2-2871
  MATLAB printer drivers 2-2856
printing
  GUIs 2-2869
  interpolated shading 2-2871
  on MS-Windows 2-2869
  with a variable file name 2-2872
  with nodisplay 2-2863
  with noFigureWindows 2-2863
  with non-normal EraseMode 2-2125 2-2725
  2-3050 2-3580 2-3685
  printing figures
    preview 2-2876
  printing tips 2-2869
  printing, suppressing 2-64
  printpreview 2-2876
  prod 2-2885
  product
    cumulative 2-798
    Kronecker tensor 2-2043
    of array elements 2-2885
    of vectors (cross) 2-785
    scalar (dot) 2-785
  profile 2-2887
  profsave 2-2894
  projection type, setting and querying 2-504
  ProjectionType, Axes property 2-322
  prompting users for input 2-1854
  prompting users to choose an item 2-2326
  propedit 2-2895 to 2-2896
  properties 2-2897
  proppanel 2-2899
  pseudoinverse 2-2794
  psi 2-2900
  publish function 2-2902
  push buttons 2-3845
  PutFullMatrix 2-2911
  pwd 2-2918

Q
qmr 2-2919
qr 2-2925
QR decomposition 2-2925
  deleting column from 2-2930
  qrdelete 2-2930
  qrinsert 2-2932
  qrupdate 2-2934
quad 2-2937
quadgk 2-2946
quadl 2-2952
quadrature 2-2937 2-2946
quadv 2-2955
quantization
performed by rgb2ind 2-3128
questdlg 2-2958
questdlg function 2-2958
quit 2-2962
quitting MATLAB 2-2962
quiver 2-2965
quiver3 2-2969
quotation mark
inserting in a string 2-1396
qz 2-2993
QZ factorization 2-2834 2-2993

R
radio buttons 2-3845
rand, RandStream method 2-2997
randi, RandStream method 2-3002
randn, RandStream method 2-3007
random
permutation 2-3008
sparse matrix 2-3356 to 2-3357
symmetric sparse matrix 2-3358
random number generators 2-2175 2-2997
2-3002 2-3007 2-3010 2-3014
randperm 2-3008
randStream
constructor 2-3014
RandStream 2-3010 2-3014
constructor 2-3010
methods
create 2-777
get 2-1517
getDefaultStream 2-1532
list 2-2175
rand 2-2997
randi 2-3002
randn 2-3007
setDefaultStream 2-3255
range space 2-2676
rank 2-3016
rank of a matrix 2-3016
RAS files
parameters that can be set when
writing 2-1826
writing 2-1819
RAS image format
specifying color order 2-1826
writing alpha data 2-1826
Raster image files
writing 2-1819
rational fraction approximation 2-3017
rbbox 2-3021 2-3065
rcond 2-3023
rdivide (M-file function equivalent for ./) 2-45
read 2-3024
readasync 2-3026
reading
binary files 2-1403
data from files 2-3706
formatted data from file 2-1421
formatted data from strings 2-3376
readme files, displaying 2-1937 2-4171
real 2-3029
real numbers 2-3029
reallog 2-3030
realmax 2-3031
realmin 2-3032
realpow 2-3033
realsqrt 2-3034
rearrange array
flip along dimension 2-1354
reverse along dimension 2-1354
rearrange matrix
flip left-right 2-1355
flip up-down 2-1356
reverse column order 2-1355
reverse row order 2-1356
RearrangeableColumn
   Uitable property 2-3993
rearranging arrays
   converting to vector 2-66
   removing first n singleton dimensions 2-3274
   removing singleton dimensions 2-3373
   reshaping 2-3114
   shifting dimensions 2-3274
   swapping dimensions 2-1924 2-2785
rearranging matrices
   converting to vector 2-66
   rotating 90\xb0 2-3160
   transposing 2-63
record 2-3035
rectangle
   properties 2-3042
   rectangle function 2-3037
rectint 2-3055
RecursionLimit
   Root property 2-3150
recycle 2-3056
reduced row echelon form 2-3171
reducespatch 2-3058
reducevolume 2-3062
reference page
   accessing from doc 2-1015
refresh 2-3065
regexprep 2-3081
regexptranslate 2-3085
registerevent 2-3088
regression
   linear 2-2835
regularly spaced vectors, creating 2-66 2-2174
rehash 2-3091
relational operators 2-50 2-2206
relational operators for handle objects 2-3095
relative accuracy
   BVP 2-478
   DDE 2-896
   norm of DDE solution 2-896
   norm of ODE solution 2-2620
   ODE 2-2620
release 2-3093
rem 2-3097
removes 2-3100
rename function 2-3102
renderer
   OpenGL 2-1265
   painters 2-1264
   zbuffer 2-1265
Renderer, Figure property 2-1264
RendererMode, Figure property 2-1268
repeatedly executing statements 2-1377 2-4175
repeatedly executing statements in parallel 2-2694
replicating a matrix 2-3103
repmat 2-3103
resample (timeseries) 2-3105
resample (tscollection) 2-3108
reset 2-3111
reshape 2-3114
residue 2-3116
residues of transfer function 2-3116
Resize, Figure property 2-1270
ResizeFcn, Figure property 2-1270
restoredefaultpath 2-3120
rethrow 2-3121
rethrow, MException method 2-3123
return 2-3125
reverse
   array along dimension 2-1354
   array dimension 2-1354
   matrix column order 2-1355
   matrix row order 2-1356
reverse Cuthill-McKee ordering 2-3635 2-3645
rewinding files to beginning of 2-1420 2-1800
RGB images
   converting to indexed 2-3127
RGB, converting to HSV 2-3126
rgb2hsv 2-3126
rgb2ind 2-3127
rgbplot 2-3130
ribbon 2-3132
right-click and context menus 2-3831
rmappdata function 2-3135
rmdir 2-3136
rmdir (ftp) function 2-3139
rmfield 2-3140
rmpath 2-3141
rmpref function 2-3142
RMS. See root-mean-square
rolling camera 2-505
root directory 2-2274
root directory for MATLAB 2-2274
Root graphics object 2-3143
root object 2-3143
root, see rootobject 2-3143
root-mean-square
  of vector 2-2570
roots 2-3154
roots of a polynomial 2-2827 to 2-2828 2-3154
rose 2-3156
Rosenbrock
  banana function 2-1367
  ODE solver 2-2609
rosser 2-3159
rot90 2-3160
rotate 2-3161
rotate3d 2-3164
rotate3d mode objects 2-3164
rotating camera 2-499
rotating camera target 2-501
Rotation, Text property 2-3698
rotations
  Jacobi 2-3358
round 2-3170
  to nearest integer 2-3170
  towards infinity 2-552
  towards minus infinity 2-1358
  towards zero 2-1353
roundoff error
  characteristic polynomial and 2-2828
  convolution theorem and 2-741
  effect on eigenvalues 2-340
  evaluating matrix functions 2-1453
  in inverse Hilbert matrix 2-1920
  partial fraction expansion and 2-3117
  polynomial roots and 2-3154
  sparse matrix conversion and 2-3324
RowName
  Uitable property 2-3993
RowStriping
  Uitable property 2-3993
rref 2-3171
rrefmovie 2-3171
rsf2csf 2-3173
rubberband box 2-3021
run 2-3175
Runge-Kutta ODE solvers 2-2608
running average 2-1320
S
save 2-3176 2-3184
  serial port I/O 2-2609
saveas 2-3188
saveobj 2-3192
savepath 2-3194
saving
  ASCII data 2-3176
  session to a file 2-977
  workspace variables 2-3176
scalar product (of vectors) 2-785
scaled complementary error function
  (defined) 2-1079
scatter 2-3195
scatter3 2-3198
scattered data, aligning
  multi-dimensional 2-2492
  two-dimensional 2-1585
scattergroup
  properties 2-3201
Schmidt semi-normalized Legendre functions 2-2076
schur 2-3219
Schur decomposition 2-3219
Schur form of matrix 2-3173 2-3219
screen, paging 2-1656
ScreenDepth, Root property 2-3150
ScreenPixelsPerInch, Root property 2-3151
ScreenSize, Root property 2-3151
script 2-3222
scrolling screen 2-1656
search path 2-3141
  adding directories to 2-126
  MATLAB 2-2739
  modifying 2-2743
  user directory 2-4061
  viewing 2-2743
search, string 2-1339
sec 2-3223
secant 2-3223
  hyperbolic 2-3226
  inverse 2-245
  inverse hyperbolic 2-248
secd 2-3225
sech 2-3226
Selected
  areasereseries property 2-233
  Axes property 2-323
  barseries property 2-368
  contour property 2-726
  errorbar property 2-1102
  Figure property 2-1271
  hggroup property 2-1683
  hgtransform property 2-1712
  Image property 2-1781
  Light property 2-2106
  Line property 2-2129
  lineseries property 2-2143
  Patch property 2-2735
  quivergroup property 2-2984
  rectangle property 2-3053
  Root property 2-3151
  scatter property 2-3213
  stairseries property 2-3398
  stem property 2-3433
  Surface property 2-3587
  surfaceplot property 2-3611
  Text property 2-3698
  Uicontrol property 2-3866
  Uitable property 2-3994
selecting areas 2-3021
SelectionHighlight
  areasereseries property 2-233
  Axes property 2-323
  barseries property 2-368
  contour property 2-727
  errorbar property 2-1103
  Figure property 2-1271
  hggroup property 2-1683
  hgtransform property 2-1712
  Image property 2-1781
  Light property 2-2106
  Line property 2-2129
  lineseries property 2-2143
  Patch property 2-2735
  quivergroup property 2-2985
  rectangle property 2-3053
  scatter property 2-3213
  stairseries property 2-3398
  stem property 2-3433
  Surface property 2-3587
  surfaceplot property 2-3611
  Text property 2-3698
  Uicontrol property 2-3866
  Uitable property 2-3994
SelectionType, Figure property 2-1271
selectmoveresize 2-3228
semicolon (special characters) 2-64
sendmail 2-3232
Separator
  Uipushtool property 2-3941
  Uitoggletool property 2-4008
Separator, Uimenu property 2-3907
sequence of matrix names (M1 through M12)
  generating 2-1115
serial 2-3234
serialbreak 2-3237
server (FTP)
  connecting to 2-1435
server variable 2-1201
session
  saving 2-977
set 2-3238 2-3242
  serial port I/O 2-3245
  timer object 2-3247
set (timeseries) 2-3250
set (tscollection) 2-3251
set hgsetget class method 2-3243
set operations
  difference 2-3256
  exclusive or 2-3270
  intersection 2-1910
  membership 2-1970
  union 2-4024
  unique 2-4026
setabstime (timeseries) 2-3252
setabstime (tscollection) 2-3253
setappdata 2-3254
setDefaultStream, RandStream method 2-3255
setdiff 2-3256
setdisp hgsetget class method 2-3257
setenv 2-3258
setfield 2-3260
setinterpmethod 2-3262
setpixelposition 2-3264
setpref function 2-3267
setstr 2-3268
settimeseriesnames 2-3269
setxor 2-3270
shading 2-3271
shading colors in surface plots 2-3271
shared libraries
  MATLAB functions
    calllib 2-487
    libfunctions 2-2085
    libfunctionsview 2-2086
    libisloaded 2-2087
    libpointer 2-2089
    libstruct 2-2091
    loadlibrary 2-2192
    unloadlibrary 2-4032
shell script 2-3653 2-4029
shiftdim 2-3274
shifting array
  circular 2-597
ShowArrowHead
  quivergroup property 2-2985
ShowBaseLine
  barseries property 2-368
ShowHiddenHandles, Root property 2-3152
showplottool 2-3275
ShowText
  contour property 2-727
shrinkfaces 2-3277
shutdown 2-2962
sign 2-3281
signum function 2-3281
simplex search 2-1369
Simpson’s rule, adaptive recursive 2-2939
Simulink
  version number, comparing 2-4100
  version number, displaying 2-4094
sin 2-3282
sind 2-3284
sine 2-3282
  hyperbolic 2-3286
  inverse 2-250
  inverse hyperbolic 2-253
single 2-3285
single quote (special characters) 2-63
singular value
    decomposition 2-3016 2-3624
    largest 2-2570
    rank and 2-3016
sinh 2-3286
size
    array dimensions 2-3288
    serial port I/O 2-3293
size (timeseries) 2-3294
size (tscollection) 2-3296
size of array dimensions 2-3288
size of fonts, see also FontSize property 2-3701
size vector 2-3114
SizeData
    scatter property 2-3214
skipping bytes (during file I/O) 2-1455
slice 2-3297
slice planes, contouring 2-735
sliders 2-3845
SliderStep, Uicontrol property 2-3866
smallest array elements 2-2385
smooth3 2-3303
smoothing 3-D data 2-3303
soccer ball (example) 2-3645
solution statistics (BVP) 2-483
sort 2-3310
sorting
    array elements 2-3310
    complex conjugate pairs 2-775
    matrix rows 2-3314
sortrows 2-3314
sound 2-3317 to 2-3318
    converting vector into 2-3317 to 2-3318
    files
        reading 2-281 2-4158
        writing 2-282 2-4163
    playing 2-4156
    recording 2-4161
resampling 2-4156
sampling 2-4161
source control on UNIX platforms
    checking out files
        function 2-578
source control system
    viewing current system 2-631
source control systems
    checking in files 2-575
    undo checkout 2-4022
spalloc 2-3319
sparse 2-3320
sparse matrix
    allocating space for 2-3319
    applying function only to nonzero elements of 2-3337
    density of 2-2567
    detecting 2-2004
    diagonal 2-3325
    finding indices of nonzero elements of 2-1327
    identity 2-3336
    number of nonzero elements in 2-2567
    permuting columns of 2-673
    random 2-3356 to 2-3357
    random symmetric 2-3358
    replacing nonzero elements of with ones 2-3350
    results of mixed operations on 2-3321
    solving least squares linear system 2-2926
    specifying maximum number of nonzero elements 2-3320
    vector of nonzero elements 2-2569
    visualizing sparsity pattern of 2-3367
sparse storage
    criterion for using 2-1437
spaugment 2-3322
spconvert 2-3323
spdiags 2-3325
special characters
    descriptions 2-1654
overloading 2-65
specular 2-3335
SpecularColorReflectance
  Patch property 2-2735
  Surface property 2-3587
  surfaceplot property 2-3611
SpecularExponent
  Patch property 2-2736
  Surface property 2-3588
  surfaceplot property 2-3612
SpecularStrength
  Patch property 2-2736
  Surface property 2-3588
  surfaceplot property 2-3612
speye 2-3336
spfun 2-3337
sph2cart 2-3339
sphere 2-3340
spherical coordinates
  defining a Light position in 2-2108
spherical coordinates 2-3339
spinmap 2-3342
spline 2-3343
spline interpolation (cubic)
  one-dimensional 2-1887 2-1897 2-1900
  2-1903
Spline Toolbox 2-1892
spones 2-3350
spparms 2-3351
sprand 2-3356
sprandn 2-3357
sprandsym 2-3358
sprank 2-3359
spreadsheets
  loading WK1 files 2-4193
  loading XLS files 2-4207
  reading into a matrix 2-1004
  writing from matrix 2-4195
  writing matrices into 2-1008
sprintf 2-3360
sqrt 2-3369
sqrtm 2-3370
square root
  of a matrix 2-3370
  of array elements 2-3369
  of real numbers 2-3034
squeeze 2-3373
sscanf 2-3376
stack, displaying 2-863
standard deviation 2-3407
start
  timer object 2-3403
startat
  timer object 2-3404
startup 2-3406
  directory and path 2-4061
startup file 2-3406
startup files 2-2273
State
  Uitoggletool property 2-4008
static text 2-3845
std 2-3407
std (timeseries) 2-3409
stem 2-3411
stem3 2-3417
step size (DDE)
  initial step size 2-900
  upper bound 2-901
step size (ODE) 2-899 2-2625
  initial step size 2-2625
  upper bound 2-2625
stop
  timer object 2-3439
stopasync 2-3440
stopwatch timer 2-3735
storage
  allocated for nonzero entries (sparse) 2-2588
  sparse 2-3320
storage allocation 2-4231
str2cell 2-568
str2double 2-3441
str2func 2-3442
str2mat 2-3446
str2num 2-3447
strcat 2-3451
stream lines
  computing 2-D 2-3457
  computing 3-D 2-3459
  drawing 2-3461
stream2 2-3457
stream3 2-3459
stretch-to-fill 2-291
strfind 2-3489
string
  comparing one to another 2-3453 2-3495
  converting from vector to 2-574
  converting matrix into 2-2264 2-2584
  converting to lowercase 2-2217
  converting to numeric array 2-3447
  converting to uppercase 2-4054
  dictionary sort of 2-3314
  finding first token in 2-3507
  searching and replacing 2-3506
  searching for 2-1339
String
  Text property 2-3698
  textarrow property 2-196
  textbox property 2-208
  Uicontrol property 2-3867
string matrix to cell array conversion 2-568
strings 2-3491
  converting to matrix (formatted) 2-3376
  inserting a quotation mark in 2-1396
  writing data to 2-3360
strjust 2-3493
strmatch 2-3494
strread 2-3498
strrep 2-3506
strtok 2-3507
strtrim 2-3510
struct 2-3511
struct2cell 2-3516
structfun 2-3517
structure array
  getting contents of field of 2-1535
  remove field from 2-3140
  setting contents of a field of 2-3260
structure arrays
  field names of 2-1231
structures
  dynamic fields 2-64
strvcat 2-3520
Style
  Light property 2-2106
  Uicontrol property 2-3870
sub2ind 2-3522
subfunction 2-1441
subplot 2-3524
subplots
  assymetrical 2-3529
  suppressing ticks in 2-3532
subsasgn 1-77 2-3537
subscripts
  in axis title 2-3753
  in text strings 2-3702
subsindex 2-3539
subspace 2-3540
subsref 2-3541
subsref (M-file function equivalent for
  A(i,j,k...)) 2-65
substruct 2-3543
subtraction (arithmetic operator) 2-40
subvolume 2-3545
sum 2-3548
  cumulative 2-800
  of array elements 2-3548
sum (timeseries) 2-3551
superiorto 2-3553
superscripts
  in axis title 2-3753
in text strings 2-3702
support 2-3554
surf2patch 2-3561
surface 2-3563
Surface
and contour plotter 2-1178
converting to a patch 2-3561
creating 2-3563
defining default properties 2-3040 2-3567
plotting mathematical functions 2-1174
properties 2-3568 2-3591
surface normals, computing for volumes 2-1984
surf1 2-3618
surfnorm 2-3622
svd 2-3624
svds 2-3627
swapbytes 2-3630
switch 2-3632
symamd 2-3634
symbfact 2-3638
symbols
operators 2-1654
symbols in text 2-196 2-208 2-3699
symmlq 2-3640
symrcm 2-3645
synchronize 2-3648
syntax 2-1655
syntax, command 2-3650
syntax, function 2-3650
syntaxes
of M-file functions, defining 2-1441
system 2-3653
UNC pathname error 2-3654
system directory, temporary 2-3663

T

table lookup. See interpolation
Tag
areaseries property 2-233

Axes property 2-323
barseries property 2-369
contour property 2-727
errorbar property 2-1103
Figure property 2-1273
hggroup property 2-1683
htransform property 2-1713
Image property 2-1781
Light property 2-2106
Line property 2-2130
lineseries property 2-2144
Patch property 2-2736
quivergroup property 2-2985
rectangle property 2-3053
Root property 2-3152
scatter property 2-3214
stairseries property 2-3399
stem property 2-3433
Surface property 2-3588
surfaceplot property 2-3612
Text property 2-3703
Uicontextmenu property 2-3840
Uicontrol property 2-3870
Uimenu property 2-3907
Uipushtool property 2-3941
Uitable property 2-3994
Uitoggletool property 2-4008
Uitoolbar property 2-4018
Tagged Image File Format (TIFF)
writing 2-1819
tan 2-3656
tand 2-3658
tangent 2-3656
fouri-quadrant, inverse 2-261
hyperbolic 2-3659
inverse 2-259
inverse hyperbolic 2-264
tanh 2-3659
tar 2-3661
target, of camera 2-506
tcpip 2-4056
tempdir 2-3663
tempname 2-3664
temporary
  files 2-3664
  system directory 2-3663
tensor, Kronecker product 2-2043
terminating MATLAB 2-2962
test matrices 2-1467
test, logical. See logical tests and detecting
tetrahedron
  mesh plot 2-3665
tetramesh 2-3665
TeX commands in text 2-196 2-208 2-3699
text 2-3670
  editing 2-2811
  subscripts 2-3702
  superscripts 2-3702
Text
  creating 2-3670
  defining default properties 2-3673
  fixed-width font 2-3687
  properties 2-3675
text mode for opened files 2-1371
TextBackgroundColor
textarrow property 2-198
TextColor
textarrow property 2-198
TextEdgeColor
textarrow property 2-198
TextLineWidth
textarrow property 2-199
TextList
contour property 2-728
TextListMode
contour property 2-728
TextMargin
textarrow property 2-199
textread 2-3706
TextRotation, textarrow property 2-199
textscan 2-3712
TextStep
  contour property 2-729
TextStepMode
  contour property 2-729
textriap 2-3726
Time,
  MException method 2-3733
throwAsCaller, MException method 2-3733
TickDir, Axes property 2-324
TickDirMode, Axes property 2-324
TickLength, Axes property 2-324
TIFF
  compression 2-1827
  encoding 2-1822
  ImageDescription field 2-1827
  maxvalue 2-1822
  parameters that can be set when
    writing 2-1826
  resolution 2-1827
  writemode 2-1827
  writing 2-1819
TIFF image format
  specifying color space 2-1826
tiling (copies of a matrix) 2-3103
time
  CPU 2-776
  elapsed (stopwatch timer) 2-3735
  required to execute commands 2-1111
time and date functions 2-1073
timer
  properties 2-3738
  timer object 2-3738
timerfind
timer object 2-3745
timerfindall
timer object 2-3747
times (M-file function equivalent for .*) 2-45
timeseries 2-3749
timestamp 2-982
title 2-3752
   with superscript 2-3753
Title, Axes property 2-325
todatenum 2-3755
toeplitz 2-3756
Toeplitz matrix 2-3756
toggle buttons 2-3845
token 2-3507
   See also string
Toolbar
   Figure property 2-1273
Toolbox
   Spline 2-1892
toolbox directory, path 2-3757
toolboxdir 2-3757
TooltipString
   Uicontrol property 2-3870
   Uipushtool property 2-3941
   Uitable property 2-3994
   Uitoggletool property 2-4008
trace 2-3758
trace of a matrix 2-973 2-3758
trailing blanks
   removing 2-908
transform
   hgtransform function 2-1691
transform, Fourier
   discrete, n-dimensional 2-1212
   discrete, one-dimensional 2-1206
   discrete, two-dimensional 2-1211
   inverse, n-dimensional 2-1749
   inverse, one-dimensional 2-1745
   inverse, two-dimensional 2-1747
   shifting the zero-frequency component of 2-1215
transformation
   See also conversion 2-538
transformations
   elementary Hermite 2-1496
   transmitting file to FTP server 2-2458
transpose
   array (arithmetic operator) 2-42
   matrix (arithmetic operator) 2-42
   transpose (M-file function equivalent for \q) 2-46
   transpose (timeseries) 2-3759
   trapz 2-3761
treelayout 2-3763
treeplot 2-3764
triangulation
   2-D plot 2-3770
   tricubic interpolation 2-1585
   tril 2-3766
   trilinear interpolation 2-1585
   trimesh 2-3767
   triple integral
      numerical evaluation 2-3768
   triplequad 2-3768
   trilplot 2-3770
   trisurf 2-3784
   triu 2-3785
   true 2-3786
   truth tables (for logical operations) 2-52
   try 2-3787
tscollection 2-3790
tsdatal.event 2-3793
tsearch 2-3794
tsearchn 2-3795
tsprops 2-3796
tstool 2-3802
type 2-3803
Type
   areaseries property 2-234
   Axes property 2-325
   barseries property 2-369
   contour property 2-729
   errorbar property 2-1103
   Figure property 2-1274
   hggroup property 2-1684
   hgtransform property 2-1713
UicontrolProperty 2-3870
Uimenu property 2-3907
Uipushtool property 2-3941
Uitable property 2-3995
Uitoggletool property 2-4009
Uitoolbar property 2-4019

typecast 2-3804

U

UData
  errorbar property 2-1104
  quivergroup property 2-2987

UDataSource
  errorbar property 2-1104
  quivergroup property 2-2987

Uibuttongroup
  defining default properties 2-3813
  uibuttongroup function 2-3808

Uibuttongroup Properties 2-3813

Uicontrol 2-3842

Uicontrol
  defining default properties 2-3848
  fixed-width font 2-3858
  types of 2-3842

Uicontrol Properties 2-3848

uicontrols
  printing 2-2869

uigetdir 2-3874

uigetfile 2-3879

uigetpref function 2-3890

uiimport 2-3894

uimenu 2-3895

Uimenu
  creating 2-3895
Index

defining default properties 2-3897
Properties 2-3897
Uimenu Properties 2-3897
uint16 2-3908
uint32 2-3908
uint64 2-3908
uint8 2-3908
uiopen 2-3910
Uipanel
    defining default properties 2-3915
uipanel function 2-3912
Uipanel Properties 2-3915
uipushtool 2-3931
Uipushtool
    defining default properties 2-3933
Uipushtool Properties 2-3933
uiputfile 2-3943
uiresume 2-3953
uisave 2-3955
uisetcolor function 2-3957
uisetfont 2-3958
uisetpref function 2-3960
uistack 2-3961
Uitable
    defining default properties 2-3968
    fixed-width font 2-3987
uitable function 2-3962
Uitable Properties 2-3968
uittoggletool 2-3997
Uittoggletool
    defining default properties 2-3999
Uittoggletool Properties 2-3999
uicontrolbar 2-4010
Uicontrolbar
    defining default properties 2-4012
Uicontrolbar Properties 2-4012
uiwait 2-4020
uminus (M-file function equivalent for unary
\xd0 ) 2-45
UNC pathname error and dos 2-1024
UN path error and system 2-3654
unconstrained minimization 2-1365
undefined numerical results 2-2481
undocheckout 2-4022
unicode2native 2-4023
unimodular matrix 2-1496
union 2-4024
unique 2-4026
unitary matrix (complex) 2-2925
Units
    annotation ellipse property 2-183
    annotation rectangle property 2-190
    arrow property 2-175
    Axes property 2-326
    doublearrow property 2-180
    Figure property 2-1274
    line property 2-186
    Root property 2-3152
    Text property 2-3703
    textarrow property 2-199
    textbox property 2-210
    Uicontrol property 2-3871
    Uitable property 2-3871
unix 2-4029
UNIX
    Web browser 2-1018
unloadlibrary 2-4032
unlocking M-files 2-2478
unmkpp 2-4037
unregisterallevents 2-4038
unregisterevent 2-4041
untar 2-4045
unwrap 2-4047
unzip 2-4052
up vector, of camera 2-508
updating figure during M-file execution 2-1029
uplus (M-file function equivalent for unary
+ ) 2-45
upper 2-4054
upper triangular matrix 2-3785
uppercase to lowercase 2-2217
urlread 2-4055
urlwrite 2-4057
usejava 2-4059
user input
  from a button menu 2-2326
UserData
  areaseries property 2-234
  Axes property 2-327
  barseries property 2-370
  contour property 2-730
  errorbar property 2-1105
  Figure property 2-1275
  hggroup property 2-1684
  hgtransform property 2-1714
  Image property 2-1782
  Light property 2-2107
  Line property 2-2130
  lineseries property 2-2145
  Patch property 2-2737
  quivergroup property 2-2986
  rectangle property 2-3054
  Root property 2-3153
  scatter property 2-3215
  stairseries property 2-3400
  stem property 2-3434
  Surface property 2-3589
  surfaceplot property 2-3613
  Text property 2-3704
  Uicontextmenu property 2-3841
  Uicontrol property 2-3871
  Uimenu property 2-3907
  Uipushtool property 2-3942
  Uitable property 2-3996
  Uitoggletool property 2-4009
  Uitoolbar property 2-4019
userpath 2-4061

\textbf{V}

\begin{itemize}
  \item validateattributes 2-4071
  \item validatestring 2-4079
  \item Value, Uicontrol property 2-3872
  \item vander 2-4086
  \item Vandermonde matrix 2-2837
  \item var 2-4087
  \item var (timeseries) 2-4088
  \item varargin 2-4090
  \item varargout 2-4092
  \item variable numbers of M-file arguments 2-4092
  \item variable-order solver (ODE) 2-2634
\end{itemize}

variables
  \begin{itemize}
    \item checking existence of 2-1135
    \item clearing from workspace 2-614
    \item global 2-1567
    \item in workspace 2-4197
    \item keeping some when clearing 2-620
    \item linking to graphs with \texttt{linkdata} 2-2159
    \item listing 2-4181
    \item local 2-1441 2-1567
    \item name of passed 2-1859
    \item opening 2-2642 2-2653
    \item persistent 2-2786
    \item saving 2-3176
    \item sizes of 2-4181
  \end{itemize}

\textbf{VData}

  \begin{itemize}
    \item quivergroup property 2-2987
  \end{itemize}

\textbf{VDataSource}

  \begin{itemize}
    \item quivergroup property 2-2988
  \end{itemize}

\textbf{vector}

  \begin{itemize}
    \item dot product 2-1025
    \item frequency 2-2214
    \item length of 2-2080
    \item product (cross) 2-785
  \end{itemize}

\textbf{vectorize 2-4093}

\textbf{vectorizing ODE function (BVP) 2-479}

vectors, creating
  \begin{itemize}
    \item logarithmically spaced 2-2214
  \end{itemize}
regularly spaced  2-66  2-2174
velocity vectors, plotting  2-692
ver  2-4094
verctrl function (Windows)  2-4096
verLessThan  2-4100
version  2-4102
version numbers
  comparing  2-4100
  displaying  2-4094
vertcat  2-4104
vertcat (M-file function equivalent for [  2-65
vertcat (timeseries)  2-4106
vertcat (tscollection)  2-4107
VertexNormals
  Patch property  2-2737
  Surface property  2-3589
  surfaceplot property  2-3613
VerticalAlignment, Text property  2-3704
VerticalAlignment, textarrow property  2-200
VerticalAlignment, textbox property  2-210
Vertices, Patch property  2-2737
video
  saving in AVI format  2-283
view  2-4111
  azimuth of viewpoint  2-4112
  coordinate system defining  2-4112
  elevation of viewpoint  2-4112
view angle, of camera  2-510
View, Axes property (obsolete)  2-327
viewing
  a group of object  2-497
  a specific object in a scene  2-497
viewmtx  2-4114
Visible
  areaseries property  2-235
  Axes property  2-327
  barseries property  2-370
  contour property  2-730
  errorbar property  2-1105
  Figure property  2-1275
higroup property  2-1685
higroup property  2-1685
hgtransform property  2-1714
Image property  2-1782
Light property  2-2107
Line property  2-2130
lineseries property  2-2145
Patch property  2-2737
quivergroup property  2-2986
rectangle property  2-3054
Root property  2-3153
scatter property  2-3215
stairseries property  2-3400
stem property  2-3434
Surface property  2-3589
surfaceplot property  2-3614
Text property  2-3705
Uicontextmenu property  2-3841
Uicontrol property  2-3872
Uimenu property  2-3907
Uipushtool property  2-3942
Uitable property  2-3996
Uitoggletool property  2-4009
Uitoolbar property  2-4019
visualizing
  cell array structure  2-566
  sparse matrices  2-3367
volumes
  calculating isosurface data  2-1987
  computing 2-D stream lines  2-3457
  computing 3-D stream lines  2-3459
  computing isosurface normals  2-1984
  contouring slice planes  2-735
  drawing stream lines  2-3461
  end caps  2-1977
  reducing face size in isosurfaces  2-3277
  reducing number of elements in  2-3062
voronoi  2-4126
Voronoi diagrams
  multidimensional visualization  2-4133
  two-dimensional visualization  2-4126
voronoin 2-4133

W
wait  
timer object 2-4137
waitbar 2-4138
waitfor 2-4142
waitforbuttonpress 2-4143
warndlg 2-4144
warning 2-4147
warning message (enabling, suppressing, and  
    displaying) 2-4147
waterfall 2-4151
.wav files  
    reading 2-4158
    writing 2-4163
waverecord 2-4161
wavfinfo 2-4155
wavplay 2-4156
wavread 2-4155 2-4158
wavrecord 2-4161
wavwrite 2-4163
WData  
    quivergroupproperty 2-2988
WDataSource  
    quivergroupproperty 2-2989
Web browser  
    displaying help in 2-1658
    specifying for UNIX 2-1018
weekday 2-4165
well conditioned 2-3023
what 2-4167
whatsnew 2-4171
which 2-4172
while 2-4175
white space characters, ASCII 2-2003 2-3507
whitebg 2-4179
who, whos  
    who 2-4181
wilkinson 2-4188
Wilkinson matrix 2-3329 2-4188
WindowButtonDownFcn, Figure property 2-1276
WindowButtonMotionFcn, Figure  
    property 2-1276
WindowButtonDownFcn, Figure property 2-1277
WindowKeyPressFcn, Figure property 2-1278
WindowKeyReleaseFcn, Figure property 2-1279
Windows Paintbrush files  
    writing 2-1818
WindowScrollWheelFcn, Figure property 2-1280
WindowStyle, Figure property 2-1282
winopen 2-4189
winqueryreg 2-4190
WK1 files  
    loading 2-4193
    writing from matrix 2-4195
wk1finfo 2-4192
wk1read 2-4193
wk1write 2-4195
workspace 2-4197
    changing context while debugging 2-857  
        2-882
    clearing items from 2-614
    consolidating memory 2-2678
    predefining variables 2-3406
    saving 2-3176
    variables in 2-4181
    viewing contents of 2-4197
workspace variables  
    reading from disk 2-2183
writing  
    binary data to file 2-1455
    formatted data to file 2-1391
WVisual, Figure property 2-1284
WVisualMode, Figure property 2-1287

X
x

Index-63
Index

annotation arrow property 2-176 2-180
annotation line property 2-186
textarrow property 2-201
X Windows Dump files
writing 2-1819
x-axis limits, setting and querying 2-4202
XAxisLocation, Axes property 2-327
XColor, Axes property 2-328
XData
areaseries property 2-235
barseries property 2-370
contour property 2-730
errorbar property 2-1105
Image property 2-1782
Line property 2-2131
lineseries property 2-2145
Patch property 2-2738
quivergroupproperty 2-2989
scatter property 2-3215
stairseries property 2-3400
stem property 2-3435
Surface property 2-3589
surfaceplot property 2-3614
XDataMode
areaseries property 2-235
barseries property 2-370
contour property 2-731
errorbar property 2-1105
lineseries property 2-2145
quivergroupproperty 2-2990
stairseries property 2-3401
stem property 2-3435
surfaceplot property 2-3614
XDataSource
areaseries property 2-236
barseries property 2-371
contour property 2-731
errorbar property 2-1106
lineseries property 2-2146
quivergroupproperty 2-2990
scatter property 2-3216
stairseries property 2-3401
stem property 2-3435
surfaceplot property 2-3614
XDir, Axes property 2-328
XDisplay, Figure property 2-1287
XGrid, Axes property 2-329
xlabel 2-4200
XLabel, Axes property 2-329
xlim 2-4202
XLim, Axes property 2-330
XLimMode, Axes property 2-330
XLS files
loading 2-4207
xlsfinfo 2-4205
xlsread 2-4207
xlsxwrite 2-4217
xmlread 2-4222
xmlwrite 2-4227
xor 2-4228
XOR, printing 2-228 2-363 2-720 2-1096 2-1709
  2-1778 2-2125 2-2138 2-2725 2-2979 2-3050
  2-3208 2-3393 2-3427 2-3580 2-3603 2-3685
XScale, Axes property 2-331
xslt 2-4229
XTick, Axes property 2-331
XTickLabel, Axes property 2-332
XTickLabelMode, Axes property 2-333
XTickMode, Axes property 2-332
XVisual, Figure property 2-1288
XVisualMode, Figure property 2-1289
XWD files
writing 2-1819
xyz coordinates. See Cartesian coordinates

Y

Y
  annotation arrow property 2-176 2-180 2-187
textarrow property 2-201
y-axis limits, setting and querying 2-4202
YAxisLocation, Axes property 2-327
YColor, Axes property 2-328
YData
  areaseries property 2-236
  barseries property 2-371
  contour property 2-732
  errorbar property 2-1106
  Image property 2-1783
  Line property 2-2131
  lineseries property 2-2146
  Patch property 2-2738
  quivergroupproperty 2-2991
  scatter property 2-3216
  stairseries property 2-3402
  stem property 2-3436
  Surface property 2-3590
  surfaceplot property 2-3615
YDataMode
  contour property 2-732
  quivergroupproperty 2-2991
  surfaceplot property 2-3615
YDataSource
  areaseries property 2-237
  barseries property 2-372
  contour property 2-732
  errorbar property 2-1107
  lineseries property 2-2147
  quivergroupproperty 2-2991
  scatter property 2-3217
  stairseries property 2-3402
  stem property 2-3436
  surfaceplot property 2-3615
YDir, Axes property 2-328
YGrid, Axes property 2-329
ylabel 2-4200
YLabel, Axes property 2-329
ylim 2-4202
YLim, Axes property 2-330
YLimMode, Axes property 2-330
YMinorGrid, Axes property 2-331
YScale, Axes property 2-331
YTick, Axes property 2-331
YTickLabel, Axes property 2-332
YTickLabelMode, Axes property 2-333
YTickMode, Axes property 2-332
Z
z-axis limits, setting and querying 2-4202
ZColor, Axes property 2-328
ZData
  contour property 2-733
  Line property 2-2131
  lineseries property 2-2147
  Patch property 2-2738
  quivergroupproperty 2-2992
  scatter property 2-3217
  stemseries property 2-3437
  Surface property 2-3590
  surfaceplot property 2-3616
ZDataSource
  contour property 2-733
  lineseries property 2-2147
  scatter property 2-3217
  surfaceplot property 2-3616
ZDir, Axes property 2-328
zero of a function, finding 2-1461
zeros 2-4231
ZGrid, Axes property 2-329
Ziggurat 2-3010
zip 2-4233
zlabel 2-4200
zlim 2-4202
ZLim, Axes property 2-330
ZLimMode, Axes property 2-330
ZMinorGrid, Axes property 2-331
zoom 2-4236
zoom mode objects 2-4237
ZScale, Axes property 2-331
ZTick, Axes property 2-331
ZTickLabel, Axes property 2-332
ZTickLabelMode, Axes property 2-333
ZTickMode, Axes property 2-332