<table>
<thead>
<tr>
<th>Revision History</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 1996</td>
</tr>
<tr>
<td>June 1997</td>
</tr>
<tr>
<td>October 1997</td>
</tr>
<tr>
<td>January 1999</td>
</tr>
<tr>
<td>June 1999</td>
</tr>
<tr>
<td>June 2001</td>
</tr>
<tr>
<td>July 2002</td>
</tr>
<tr>
<td>June 2004</td>
</tr>
<tr>
<td>September 2006</td>
</tr>
</tbody>
</table>
# Functions — By Category

## 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 ................................................. 1-19
- Elementary Math .............................................. 1-23
- Polynomials ................................................... 1-28
- Interpolation and Computational Geometry .............. 1-28
- Cartesian Coordinate System Conversion .................. 1-31
- Nonlinear Numerical Methods ............................... 1-31
- Specialized Math .............................................. 1-35
- Sparse Matrices .............................................. 1-35
- Math Constants .............................................. 1-39

## Data Analysis
- Basic Operations .............................................. 1-41
- Descriptive Statistics ....................................... 1-41
- Filtering and Convolution .................................. 1-42
- Interpolation and Regression ............................... 1-42
- Fourier Transforms ......................................... 1-43
- Derivatives and Integrals .................................. 1-43
- Time Series Objects ......................................... 1-44
- Time Series Collections .................................... 1-47

## Programming and Data Types
- Data Types ................................................... 1-49
- Data Type Conversion ....................................... 1-58
- Operators and Special Characters ......................... 1-60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String Functions</strong></td>
<td>1-62</td>
</tr>
<tr>
<td><strong>Bit-wise Functions</strong></td>
<td>1-65</td>
</tr>
<tr>
<td><strong>Logical Functions</strong></td>
<td>1-66</td>
</tr>
<tr>
<td><strong>Relational Functions</strong></td>
<td>1-66</td>
</tr>
<tr>
<td><strong>Set Functions</strong></td>
<td>1-67</td>
</tr>
<tr>
<td><strong>Date and Time Functions</strong></td>
<td>1-67</td>
</tr>
<tr>
<td><strong>Programming in MATLAB</strong></td>
<td>1-68</td>
</tr>
<tr>
<td><strong>File I/O</strong></td>
<td>1-75</td>
</tr>
<tr>
<td>File Name Construction</td>
<td>1-75</td>
</tr>
<tr>
<td>Opening, Loading, Saving Files</td>
<td>1-76</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>1-76</td>
</tr>
<tr>
<td>Low-Level File I/O</td>
<td>1-76</td>
</tr>
<tr>
<td>Text Files</td>
<td>1-77</td>
</tr>
<tr>
<td>XML Documents</td>
<td>1-78</td>
</tr>
<tr>
<td>Spreadsheets</td>
<td>1-78</td>
</tr>
<tr>
<td>Scientific Data</td>
<td>1-79</td>
</tr>
<tr>
<td>Audio and Audio/Video</td>
<td>1-80</td>
</tr>
<tr>
<td>Images</td>
<td>1-82</td>
</tr>
<tr>
<td>Internet Exchange</td>
<td>1-83</td>
</tr>
<tr>
<td><strong>Graphics</strong></td>
<td>1-85</td>
</tr>
<tr>
<td>Basic Plots and Graphs</td>
<td>1-85</td>
</tr>
<tr>
<td>Plotting Tools</td>
<td>1-86</td>
</tr>
<tr>
<td>Annotating Plots</td>
<td>1-86</td>
</tr>
<tr>
<td>Specialized Plotting</td>
<td>1-87</td>
</tr>
<tr>
<td>Bit-Mapped Images</td>
<td>1-91</td>
</tr>
<tr>
<td>Printing</td>
<td>1-91</td>
</tr>
<tr>
<td>Handle Graphics</td>
<td>1-92</td>
</tr>
<tr>
<td><strong>3-D Visualization</strong></td>
<td>1-96</td>
</tr>
<tr>
<td>Surface and Mesh Plots</td>
<td>1-96</td>
</tr>
<tr>
<td>View Control</td>
<td>1-98</td>
</tr>
<tr>
<td>Lighting</td>
<td>1-100</td>
</tr>
<tr>
<td>Transparency</td>
<td>1-100</td>
</tr>
<tr>
<td>Volume Visualization</td>
<td>1-101</td>
</tr>
<tr>
<td><strong>Creating Graphical User Interfaces</strong></td>
<td>1-103</td>
</tr>
<tr>
<td>Predefined Dialog Boxes</td>
<td>1-103</td>
</tr>
<tr>
<td>Deploying User Interfaces</td>
<td>1-104</td>
</tr>
<tr>
<td>Developing User Interfaces</td>
<td>1-104</td>
</tr>
<tr>
<td>User Interface Objects</td>
<td>1-105</td>
</tr>
</tbody>
</table>
External Interfaces .......................... 1-108
  Dynamic Link Libraries ..................... 1-108
  Java ............................................ 1-109
  Component Object Model and ActiveX ....... 1-110
  Dynamic Data Exchange ...................... 1-112
  Web Services .................................. 1-113
  Serial Port Devices ......................... 1-113

Functions — Alphabetical List

Index
## Functions — By Category

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop Tools and Development Environment</td>
<td>Startup, Command Window, help, editing and debugging, tuning, other general functions</td>
</tr>
<tr>
<td>Mathematics</td>
<td>Arrays and matrices, linear algebra, other areas of mathematics</td>
</tr>
<tr>
<td>Data Analysis</td>
<td>Basic data operations, descriptive statistics, covariance and correlation, filtering and convolution, numerical derivatives and integrals, Fourier transforms, time series analysis</td>
</tr>
<tr>
<td>Programming and Data Types</td>
<td>Function/expression evaluation, program control, function handles, object oriented programming, error handling, operators, data types, dates and times, timers</td>
</tr>
<tr>
<td>File I/O</td>
<td>General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images</td>
</tr>
<tr>
<td>Graphics</td>
<td>Line plots, annotating graphs, specialized plots, images, printing, Handle Graphics</td>
</tr>
<tr>
<td>3-D Visualization</td>
<td>Surface and mesh plots, view control, lighting and transparency, volume visualization</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>Creating Graphical User Interfaces (p. 1-103)</td>
<td>GUIDE, programming graphical user interfaces</td>
</tr>
<tr>
<td>External Interfaces (p. 1-108)</td>
<td>Interfaces to DLLs, Java, COM and ActiveX, DDE, Web services, and serial port devices, and C and Fortran routines</td>
</tr>
</tbody>
</table>
# Desktop Tools and Development Environment

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup and Shutdown (p. 1-3)</td>
<td>Startup and shutdown options, preferences</td>
</tr>
<tr>
<td>Command Window and History (p. 1-4)</td>
<td>Control Command Window and History, enter statements and run functions</td>
</tr>
<tr>
<td>Help for Using MATLAB (p. 1-5)</td>
<td>Command line help, online documentation in the Help browser, demos</td>
</tr>
<tr>
<td>Workspace, Search Path, and File Operations (p. 1-6)</td>
<td>Work with files, MATLAB search path, manage variables</td>
</tr>
<tr>
<td>Programming Tools (p. 1-8)</td>
<td>Edit and debug M-files, improve performance, source control, publish results</td>
</tr>
<tr>
<td>System (p. 1-11)</td>
<td>Identify current computer, license, product version, and more</td>
</tr>
</tbody>
</table>

## Startup and Shutdown

- **exit** Terminate MATLAB (same as `quit`)
- **finish** MATLAB termination M-file
- **matlab (UNIX)** Start MATLAB (UNIX systems)
- **matlab (Windows)** Start MATLAB (Windows systems)
- **matlabrc** MATLAB startup M-file for single-user systems or system administrators
- **prefdir** Directory containing preferences, history, and layout files
- **preferences** Open Preferences dialog box for MATLAB and related products
quit

Terminate MATLAB

startup

MATLAB startup M-file for user-defined options

**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><code>builddocsearchdb</code></td>
<td>Build searchable documentation database</td>
</tr>
<tr>
<td><code>demo</code></td>
<td>Access product demos via Help browser</td>
</tr>
<tr>
<td><code>doc</code></td>
<td>Reference page in Help browser</td>
</tr>
<tr>
<td><code>docopt</code></td>
<td>Web browser for UNIX platforms</td>
</tr>
<tr>
<td><code>docsearch</code></td>
<td>Open Help browser Search pane and search for specified term</td>
</tr>
<tr>
<td><code>echodemo</code></td>
<td>Run M-file demo step-by-step in Command Window</td>
</tr>
<tr>
<td><code>help</code></td>
<td>Help for MATLAB functions in Command Window</td>
</tr>
<tr>
<td><code>helpbrowser</code></td>
<td>Open Help browser to access all online documentation and demos</td>
</tr>
<tr>
<td><code>helpwin</code></td>
<td>Provide access to M-file help for all functions</td>
</tr>
<tr>
<td><code>info</code></td>
<td>Information about contacting The MathWorks</td>
</tr>
<tr>
<td><code>lookfor</code></td>
<td>Search for keyword in all help entries</td>
</tr>
<tr>
<td><code>playshow</code></td>
<td>Run M-file demo (deprecated; use <code>echodemo</code> instead)</td>
</tr>
<tr>
<td><code>support</code></td>
<td>Open MathWorks Technical Support Web page</td>
</tr>
<tr>
<td><code>web</code></td>
<td>Open Web site or file in Web browser or Help browser</td>
</tr>
<tr>
<td><code>whatsnew</code></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 class
openvar  Open workspace variable in Array Editor or other tool for graphical editing
pack  Consolidate workspace memory
uiimport  Open Import Wizard to import data
which  Locate functions and files
workspace  Open Workspace browser to manage workspace

**Search Path**

addpath  Add directories to MATLAB search path
genpath  Generate path string
partialpath  Partial pathname description
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>View or change MATLAB directory search path</td>
</tr>
<tr>
<td>path2rc</td>
<td>Save current MATLAB search path to pathdef.m file</td>
</tr>
<tr>
<td>pathdef</td>
<td>Directories in MATLAB search path</td>
</tr>
<tr>
<td>pathsep</td>
<td>Path separator for current platform</td>
</tr>
<tr>
<td>pathtool</td>
<td>Open Set Path dialog box to view and change MATLAB path</td>
</tr>
<tr>
<td>restoredefaultpath</td>
<td>Restore default MATLAB search path</td>
</tr>
<tr>
<td>rmpath</td>
<td>Remove directories from MATLAB search path</td>
</tr>
<tr>
<td>savepath</td>
<td>Save current MATLAB search path to pathdef.m file</td>
</tr>
</tbody>
</table>

### File Operations

See also “File I/O” on page 1-75 functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd</td>
<td>Change working directory</td>
</tr>
<tr>
<td>copyfile</td>
<td>Copy file or directory</td>
</tr>
<tr>
<td>delete</td>
<td>Remove files or graphics objects</td>
</tr>
<tr>
<td>dir</td>
<td>Directory listing</td>
</tr>
<tr>
<td>exist</td>
<td>Check existence of variable, function, directory, or Java class</td>
</tr>
<tr>
<td>fileattrib</td>
<td>Set or get attributes of file or directory</td>
</tr>
<tr>
<td>filebrowser</td>
<td>Current Directory browser</td>
</tr>
<tr>
<td>isdir</td>
<td>Determine whether input is a directory</td>
</tr>
<tr>
<td>lookfor</td>
<td>Search for keyword in all help entries</td>
</tr>
</tbody>
</table>
ls
Directory contents on UNIX system
matlabroot
Root directory of MATLAB installation
mkdir
Make new directory
movefile
Move file or directory
pwd
Identify current directory
recycle
Set option to move deleted files to recycle folder
rehash
Refresh function and file system path caches
rmdir
Remove directory
toolboxdir
Root directory for specified toolbox
type
Display contents of file
web
Open Web site or file in Web browser or Help browser
what
List MATLAB files in current directory
which
Locate functions and files

Programming Tools

Edit and Debug M-Files (p. 1-9)
Edit and debug M-files
Improve Performance and Tune M-Files (p. 1-9)
Improve performance and find potential problems in M-files
Source Control (p. 1-10)
Interface MATLAB with source control system
Publishing (p. 1-10)
Publish M-file code and results
**Edit and Debug M-Files**

clipboard
- Copy and paste strings to and from system clipboard

datatipinfo
- Produce short description of input variable

dbclear
- Clear breakpoints

dbcont
- Resume execution

dbdowm
- Change local workspace context when in debug mode

dbquit
- Quit debug mode

dbstack
- Function call stack

dbstatus
- List all breakpoints

dbstep
- Execute one or more lines from current breakpoint

dbstopt
- Set breakpoints

dbtype
- List M-file with line numbers

dbup
- Change local workspace context

dbsert
- List M-file debugging functions

edit
- Edit or create M-file

keyboard
- Input from keyboard

**Improve Performance and Tune M-Files**

memory
- Help for memory limitations

mlint
- Check M-files for possible problems

mlinptrpt
- Run mlint for file or directory, reporting results in browser

pack
- Consolidate workspace memory

profile
- Profile execution time for function
Functions — By Category

profsave
Save profile report in HTML format

rehash
Refresh function and file system path caches

sparse
Create sparse matrix

zeros
Create array of all zeros

Source Control

checkin
Check files into source control system (UNIX)

checkout
Check files out of source control system (UNIX)

cmopts
Name of source control system

customverctrl
Allow custom source control system (UNIX)

undocheckout
Undo previous checkout from source control system (UNIX)

verctrl
Source control actions (Windows)

Publishing

grabcode
MATLAB code from M-files published to HTML

notebook
Open M-book in Microsoft Word (Windows)

publish
Publish M-file containing cells, saving output to file of specified type
**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 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>MATLAB server host identification number</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 Microsoft 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 whether running Macintosh OS X versions of MATLAB</td>
</tr>
<tr>
<td>ispc</td>
<td>Determine whether PC (Windows) version of MATLAB</td>
</tr>
<tr>
<td>isstudent</td>
<td>Determine whether Student Version of MATLAB</td>
</tr>
<tr>
<td>isunix</td>
<td>Determine whether UNIX version of MATLAB</td>
</tr>
<tr>
<td>javachk</td>
<td>Generate error message based on 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 Java feature is supported in MATLAB</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. 1-19)</td>
<td>Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization</td>
</tr>
<tr>
<td>Elementary Math (p. 1-23)</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. 1-28)</td>
<td>Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation</td>
</tr>
<tr>
<td>Cartesian Coordinate System Conversion (p. 1-31)</td>
<td>Conversions between Cartesian and polar or spherical coordinates</td>
</tr>
<tr>
<td>Specialized Math (p. 1-35)</td>
<td>Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions</td>
</tr>
<tr>
<td>Sparse Matrices (p. 1-35)</td>
<td>Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations</td>
</tr>
<tr>
<td>Math Constants (p. 1-39)</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. 1-17)
Create, sort, rotate, permute, reshape, and shift array contents

Specialized Matrices (p. 1-18)
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

isequalwithnan
Test arrays for equality, treating NaNs as equal

isfinite
Array elements that are finite

isfloat
Determine whether input is floating-point array

isin
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)
1. ^  Array power (element-wise)
2. \  Left array divide (element-wise)
3. /  Right array divide (element-wise)

Elementary Matrices and Arrays

- **blkdiag**  Construct block diagonal matrix from input arguments
- **diag**  Diagonal matrices and diagonals of matrix
- **eye**  Identity matrix
- **freqspace**  Frequency spacing for frequency response
- **ind2sub**  Subscripts from linear index
- **linspace**  Generate linearly spaced vectors
- **logspace**  Generate logarithmically spaced vectors
- **meshgrid**  Generate X and Y arrays for 3-D plots
- **ndgrid**  Generate arrays for N-D functions and interpolation
- **ones**  Create array of all ones
- **rand**  Uniformly distributed pseudorandom numbers
- **randn**  Normally distributed random numbers
- **sub2ind**  Single index from subscripts
- **zeros**  Create array of all zeros
### Array Operations

See “Linear Algebra” on page 1-19 and “Elementary Math” on page 1-23 for other array operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumarray</td>
<td>Construct array with accumulation</td>
</tr>
<tr>
<td>arrayfun</td>
<td>Apply function to each element of array</td>
</tr>
<tr>
<td>bsxfun</td>
<td>Applies element-by-element binary operation to two arrays with singleton expansion enabled</td>
</tr>
<tr>
<td>cast</td>
<td>Cast variable to different data type</td>
</tr>
<tr>
<td>cross</td>
<td>Vector cross product</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum</td>
</tr>
<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>
</tbody>
</table>
Functions — By Category

- **diag**
  - Diagonal matrices and diagonals of matrix

- **end**
  - Terminate block of code, or indicate last array index

- **flipdim**
  - Flip array along specified dimension

- **fliplr**
  - Flip matrix left to right

- **flipud**
  - Flip matrix up to down

- **horzcat**
  - Concatenate arrays horizontally

- **inline**
  - Construct inline object

- **ipermute**
  - Inverse permute dimensions of N-D array

- **permute**
  - Rearrange dimensions of N-D array

- **repmat**
  - Replicate and tile array

- **reshape**
  - Reshape array

- **rot90**
  - Rotate matrix 90 degrees

- **shiftdim**
  - Shift dimensions

- **sort**
  - Sort array elements in ascending or descending order

- **sortrows**
  - Sort rows in ascending order

- **squeeze**
  - Remove singleton dimensions

- **vectorize**
  - Vectorize expression

- **vertcat**
  - Concatenate arrays vertically

**Specialized Matrices**

- **compan**
  - Companion matrix

- **gallery**
  - Test matrices

- **hadamard**
  - Hadamard matrix

- **hankel**
  - Hankel matrix
hilb
invhilb
magic
pascal
rosser
toeplitz
vander
wilkinson

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. 1-19)
Compute norm, rank, determinant, condition number, etc.

Linear Equations (p. 1-20)
Solve linear systems, least squares, LU factorization, Cholesky factorization, etc.

Eigenvalues and Singular Values (p. 1-21)
Eigenvalues, eigenvectors, Schur decomposition, Hessenburg matrices, etc.

Matrix Logarithms and Exponentials (p. 1-22)
Matrix logarithms, exponentials, square root

Factorization (p. 1-22)
Cholesky, LU, and QR factorizations, diagonal forms, singular value decomposition

Matrix Analysis

cond
condeig
Condition number with respect to inversion
Condition number with respect to eigenvalues
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>det</td>
<td>Matrix determinant</td>
</tr>
<tr>
<td>norm</td>
<td>Vector and matrix norms</td>
</tr>
<tr>
<td>normest</td>
<td>2-norm estimate</td>
</tr>
<tr>
<td>null</td>
<td>Null space</td>
</tr>
<tr>
<td>orth</td>
<td>Range space of matrix</td>
</tr>
<tr>
<td>rank</td>
<td>Rank of matrix</td>
</tr>
<tr>
<td>rcond</td>
<td>Matrix reciprocal condition number estimate</td>
</tr>
<tr>
<td>rref</td>
<td>Reduced row echelon form</td>
</tr>
<tr>
<td>subspace</td>
<td>Angle between two subspaces</td>
</tr>
<tr>
<td>trace</td>
<td>Sum of diagonal elements</td>
</tr>
</tbody>
</table>

**Linear Equations**

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

### Eigenvalues and Singular Values

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
<td>Diagonal scaling to improve eigenvalue accuracy</td>
</tr>
<tr>
<td>cdf2rdf</td>
<td>Convert complex diagonal form to real block diagonal form</td>
</tr>
<tr>
<td>condeig</td>
<td>Condition number with respect to eigenvalues</td>
</tr>
<tr>
<td>eig</td>
<td>Find eigenvalues and eigenvectors</td>
</tr>
<tr>
<td>eigs</td>
<td>Find largest eigenvalues and eigenvectors of sparse matrix</td>
</tr>
<tr>
<td>gsvd</td>
<td>Generalized singular value decomposition</td>
</tr>
<tr>
<td>hess</td>
<td>Hessenberg form of matrix</td>
</tr>
<tr>
<td>ordeig</td>
<td>Eigenvalues of quasitriangular matrices</td>
</tr>
<tr>
<td>ordqz</td>
<td>Reorder eigenvalues in QZ factorization</td>
</tr>
<tr>
<td>ordschur</td>
<td>Reorder eigenvalues in Schur factorization</td>
</tr>
<tr>
<td>poly</td>
<td>Polynomial with specified roots</td>
</tr>
<tr>
<td>polyeig</td>
<td>Polynomial eigenvalue problem</td>
</tr>
</tbody>
</table>
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. 1-25)  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 (e.g., Prime Factors)  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
- `asec`  
  Inverse secant; result in radians
- `asecd`  
  Inverse secant; result in degrees
- `asech`  
  Inverse hyperbolic secant
- `asin`  
  Inverse sine; result in radians
- `asind`  
  Inverse sine; result in degrees
- `asinh`  
  Inverse hyperbolic sine
- `atan`  
  Inverse tangent; result in radians
- `atan2`  
  Four-quadrant inverse tangent
- `atand`  
  Inverse tangent; result in degrees
- `atanh`  
  Inverse hyperbolic tangent
- `cos`  
  Cosine of argument in radians
- `cosd`  
  Cosine of argument in degrees
- `cosh`  
  Hyperbolic cosine
- `cot`  
  Cotangent of argument in radians
- `cotd`  
  Cotangent of argument in degrees
- `coth`  
  Hyperbolic cotangent
- `csc`  
  Cosecant of argument in radians
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csch</td>
<td>Hyperbolic cosecant</td>
</tr>
<tr>
<td>cscd</td>
<td>Cosecant of argument in degrees</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>sind</td>
<td>Sine of argument in degrees</td>
</tr>
<tr>
<td>sin</td>
<td>Sine of argument in radians</td>
</tr>
<tr>
<td>sinh</td>
<td>Hyperbolic sine of argument in radians</td>
</tr>
<tr>
<td>tand</td>
<td>Tangent of argument in degrees</td>
</tr>
<tr>
<td>tan</td>
<td>Tangent of argument in radians</td>
</tr>
<tr>
<td>tanh</td>
<td>Hyperbolic tangent</td>
</tr>
</tbody>
</table>

**Exponential**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>Exponential</td>
</tr>
<tr>
<td>expm1</td>
<td>Compute $\exp(x) - 1$ accurately for small values of $x$</td>
</tr>
<tr>
<td>log</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>log10</td>
<td>Common (base 10) logarithm</td>
</tr>
<tr>
<td>log1p</td>
<td>Compute $\log(1+x)$ accurately for small values of $x$</td>
</tr>
<tr>
<td>log2</td>
<td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td>
</tr>
<tr>
<td>nextpow2</td>
<td>Next higher power of 2</td>
</tr>
<tr>
<td>nthroot</td>
<td>Real $n$th root of real numbers</td>
</tr>
<tr>
<td>pow2</td>
<td>Base 2 power and scale floating-point numbers</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>reallog</td>
<td>Natural logarithm for nonnegative real arrays</td>
</tr>
<tr>
<td>realpow</td>
<td>Array power for real-only output</td>
</tr>
<tr>
<td>realsqrt</td>
<td>Square root for nonnegative real arrays</td>
</tr>
<tr>
<td>sqrt</td>
<td>Square root</td>
</tr>
<tr>
<td></td>
<td><strong>Complex</strong></td>
</tr>
<tr>
<td>abs</td>
<td>Absolute value and complex magnitude</td>
</tr>
<tr>
<td>angle</td>
<td>Phase angle</td>
</tr>
<tr>
<td>complex</td>
<td>Construct complex data from real and imaginary components</td>
</tr>
<tr>
<td>conj</td>
<td>Complex conjugate</td>
</tr>
<tr>
<td>cplxpair</td>
<td>Sort complex numbers into complex conjugate pairs</td>
</tr>
<tr>
<td>i</td>
<td>Imaginary unit</td>
</tr>
<tr>
<td>imag</td>
<td>Imaginary part of complex number</td>
</tr>
<tr>
<td>isreal</td>
<td>Determine whether input is real array</td>
</tr>
<tr>
<td>j</td>
<td>Imaginary unit</td>
</tr>
<tr>
<td>real</td>
<td>Real part of complex number</td>
</tr>
<tr>
<td>sign</td>
<td>Signum function</td>
</tr>
<tr>
<td>unwrap</td>
<td>Correct phase angles to produce smoother phase plots</td>
</tr>
</tbody>
</table>
### Rounding and Remainder

- **ceil**: Round toward infinity
- **fix**: Round toward zero
- **floor**: Round toward minus infinity
- **idivide**: Integer division with rounding option
- **mod**: Modulus after division
- **rem**: Remainder after division
- **round**: Round to nearest integer

### Discrete Math (e.g., Prime Factors)

- **factor**: Prime factors
- **factorial**: Factorial function
- **gcd**: Greatest common divisor
- **isprime**: Array elements that are prime numbers
- **lcm**: Least common multiple
- **nchoosek**: Binomial coefficient or all combinations
- **perms**: All possible permutations
- **primes**: Generate list of prime numbers
- **rat, rats**: Rational fraction approximation
Polynomials

conv
Convolution and polynomial multiplication

deconv
Deconvolution and polynomial division

poly
Polynomial with specified roots

polyder
Polynomial derivative

polyeig
Polynomial eigenvalue problem

polyfit
Polynomial curve fitting

polyint
Integrate polynomial analytically

polyval
Polynomial evaluation

polyvalm
Matrix polynomial evaluation

residue
Convert between partial fraction expansion and polynomial coefficients

roots
Polynomial roots

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. 1-30)
Plot convex hull, plotting functions

Voronoi Diagrams (p. 1-30)
Plot Voronoi diagram, patch graphics object, plotting functions

Domain Generation (p. 1-31)
Generate arrays for 3-D plots, or for N-D functions and interpolation
# Interpolation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>griddata</td>
<td>Data gridding</td>
</tr>
<tr>
<td>griddata3</td>
<td>Data gridding and hypersurface fitting for 3-D data</td>
</tr>
<tr>
<td>griddatan</td>
<td>Data gridding and hypersurface fitting (dimension &gt;= 2)</td>
</tr>
<tr>
<td>interp1</td>
<td>1-D data interpolation (table lookup)</td>
</tr>
<tr>
<td>interp1q</td>
<td>Quick 1-D linear interpolation</td>
</tr>
<tr>
<td>interp2</td>
<td>2-D data interpolation (table lookup)</td>
</tr>
<tr>
<td>interp3</td>
<td>3-D data interpolation (table lookup)</td>
</tr>
<tr>
<td>interpft</td>
<td>1-D interpolation using FFT method</td>
</tr>
<tr>
<td>interpn</td>
<td>N-D data interpolation (table lookup)</td>
</tr>
<tr>
<td>meshgrid</td>
<td>Generate X and Y arrays for 3-D plots</td>
</tr>
<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>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

delaunay

delaunay3

delaunayn

dsearch

dsearchn

tetramesh

trimesh

triplot

trisurf

tsearch

tsearchn

Convex Hull

convhull

convhulln

patch

plot

trisurf

Voronoi Diagrams

dsearch

patch

plot
Mathematics

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 (IVP) (p. 1-32)
Solve stiff and nonstiff differential equations, define the problem, set solver options, evaluate solution

Delay Differential Equations (p. 1-33)
Solve delay differential equations with constant and general delays, set solver options, evaluate solution

Boundary Value Problems (p. 1-33)
Solve boundary value problems for ordinary differential equations, set solver options, evaluate solution
Partial Differential Equations (p. 1-34)
Solve initial-boundary value problems for parabolic-elliptic PDEs, evaluate solution

Optimization (p. 1-34)
Find minimum of single and multivariable functions, solve nonnegative least-squares constraint problem

Numerical Integration (Quadrature) (p. 1-34)
Evaluate Simpson, Lobatto, and vectorized quadratures, evaluate double and triple integrals

**Ordinary Differential Equations (IVP)**

decic
Compute consistent initial conditions for ode15i

deval
Evaluate solution of differential equation problem

ode15i
Solve fully implicit differential equations, variable order method

ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
Solve initial value problems for ordinary differential equations

odefile
Define differential equation problem for ordinary differential equation solvers

odeget
Ordinary differential equation options parameters

odeset
Create or alter options structure for ordinary differential equation solvers

odextend
Extend solution of initial value problem for ordinary differential equation
### Delay Differential Equations

- **dde23**: Solve delay differential equations (DDEs) with constant delays
- **ddeget**: Extract properties from delay differential equations options structure
- **ddesd**: Solve delay differential equations (DDEs) with general delays
- **ddeset**: Create or alter delay differential equations options structure
- **deval**: Evaluate solution of differential equation problem

### Boundary Value Problems

- **bvp4c**: 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
- **quad**
  - Numerically evaluate integral, adaptive Simpson 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
- beta: Beta function
- betainc: Incomplete beta function
- betaln: Logarithm of beta function
- ellipj: Jacobi elliptic functions
- ellipke: Complete elliptic integrals of first and second kind
- erf, erfc, erfcx, erfinv, erfcinv: Error functions
- expint: Exponential integral
- gamma, gammainc, gammainv: Gamma functions
- legendre: Associated Legendre functions
- psi: Psi (polygamma) function

Sparse Matrices

Elementary Sparse Matrices (p. 1-36)  Create random and nonrandom sparse matrices

Full to Sparse Conversion (p. 1-36)  Convert full matrix to sparse, sparse matrix to full
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with Sparse Matrices (p. 1-37)</td>
<td>Test matrix for sparseness, get information on sparse matrix, allocate sparse matrix, apply function to nonzero elements, visualize sparsity pattern.</td>
</tr>
<tr>
<td>Reordering Algorithms (p. 1-37)</td>
<td>Random, column, minimum degree, Dulmage-Mendelsohn, and reverse Cuthill-McKee permutations</td>
</tr>
<tr>
<td>Linear Algebra (p. 1-38)</td>
<td>Compute norms, eigenvalues, factorizations, least squares, structural rank</td>
</tr>
<tr>
<td>Linear Equations (Iterative Methods) (p. 1-38)</td>
<td>Methods for conjugate and biconjugate gradients, residuals, lower quartile</td>
</tr>
<tr>
<td>Tree Operations (p. 1-39)</td>
<td>Elimination trees, tree plotting, factorization analysis</td>
</tr>
</tbody>
</table>

### Elementary Sparse Matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spdiags</td>
<td>Extract and create sparse band and diagonal matrices</td>
</tr>
<tr>
<td>speye</td>
<td>Sparse identity matrix</td>
</tr>
<tr>
<td>sprand</td>
<td>Sparse uniformly distributed random matrix</td>
</tr>
<tr>
<td>sprandn</td>
<td>Sparse normally distributed random matrix</td>
</tr>
<tr>
<td>sprandsym</td>
<td>Sparse symmetric random matrix</td>
</tr>
</tbody>
</table>

### Full to Sparse Conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>Find indices and values of nonzero elements</td>
</tr>
<tr>
<td>full</td>
<td>Convert sparse matrix to full matrix</td>
</tr>
</tbody>
</table>
Working with Sparse Matrices

- **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  Find largest eigenvalues and eigenvectors of sparse 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
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

**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

**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>realmax</td>
<td>Largest positive floating-point number</td>
</tr>
<tr>
<td>realmin</td>
<td>Smallest positive floating-point number</td>
</tr>
</tbody>
</table>
Data Analysis

Basic Operations (p. 1-41)  
Sums, products, sorting

Descriptive Statistics (p. 1-41)  
Statistical summaries of data

Filtering and Convolution (p. 1-42)  
Data preprocessing

Interpolation and Regression (p. 1-42)  
Data fitting

Fourier Transforms (p. 1-43)  
Frequency content of data

Derivatives and Integrals (p. 1-43)  
Data rates and accumulations

Time Series Objects (p. 1-44)  
Methods for timeseries objects

Time Series Collections (p. 1-47)  
Methods for tscollection objects

Basic Operations

cumprod  
Cumulative product

cumsum  
Cumulative sum

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

max  
Largest elements in array

mean  
Average or mean value of array

median  
Median value of array
1 Functions — By Category

min Smallest elements in array
mode Most frequent values in array
std Standard deviation
var Variance

Filtering and Convolution

conv Convolution and polynomial multiplication
conv2 2-D convolution
convn N-D convolution
decov Deconvolution and polynomial division
detrend Remove linear trends
filter 1-D digital filter
filter2 2-D digital filter

Interpolation and Regression

interp1 1-D data interpolation (table lookup)
interp2 2-D data interpolation (table lookup)
interp3 3-D data interpolation (table lookup)
interpn N-D data interpolation (table lookup)
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  
Numerical gradient  
polyder  
Polynomial derivative  
polyint  
Integrate polynomial analytically  
trapz  
Trapezoidal numerical integration

**Time Series Objects**

**General Purpose (p. 1-44)**
Combine timeseries objects, query and set timeseries object properties, plot timeseries objects

**Data Manipulation (p. 1-45)**
Add or delete data, manipulate timeseries objects

**Event Data (p. 1-46)**
Add or delete events, create new timeseries objects based on event data

**Descriptive Statistics (p. 1-46)**
Descriptive statistics for timeseries objects

**General Purpose**

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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeseries</td>
<td>Create timeseries object</td>
</tr>
<tr>
<td>tsdata.event</td>
<td>Construct event object for timeseries object</td>
</tr>
<tr>
<td>tsprops</td>
<td>Help on timeseries object properties</td>
</tr>
<tr>
<td>tstool</td>
<td>Open Time Series Tools GUI</td>
</tr>
</tbody>
</table>

**Data Manipulation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addsample</td>
<td>Add data sample to timeseries object</td>
</tr>
<tr>
<td>ctranspose (timeseries)</td>
<td>Transpose timeseries object</td>
</tr>
<tr>
<td>delsample</td>
<td>Remove sample from timeseries object</td>
</tr>
<tr>
<td>detrend (timeseries)</td>
<td>Subtract mean or best-fit line and all NaNs from time series</td>
</tr>
<tr>
<td>filter (timeseries)</td>
<td>Shape frequency content of timeseries</td>
</tr>
<tr>
<td>getabstime (timeseries)</td>
<td>Extract date-string time vector into cell array</td>
</tr>
<tr>
<td>getinterpmethod</td>
<td>Interpolation method for timeseries object</td>
</tr>
<tr>
<td>getsampleusingtime (timeseries)</td>
<td>Extract data samples into new timeseries object</td>
</tr>
<tr>
<td>idealfilter (timeseries)</td>
<td>Apply ideal (noncausal) filter to timeseries object</td>
</tr>
<tr>
<td>resample (timeseries)</td>
<td>Select or interpolate timeseries data using new time vector</td>
</tr>
<tr>
<td>setabstime (timeseries)</td>
<td>Set times of timeseries object as date strings</td>
</tr>
<tr>
<td>setinterpmethod</td>
<td>Set default interpolation method for timeseries object</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>synchronize</code></td>
<td>Synchronize and resample two <code>timeseries</code> objects using common</td>
</tr>
<tr>
<td></td>
<td>time vector</td>
</tr>
<tr>
<td><code>transpose (timeseries)</code></td>
<td>Transpose <code>timeseries</code> object</td>
</tr>
<tr>
<td><code>vertcat (timeseries)</code></td>
<td>Vertical concatenation of <code>timeseries</code> objects</td>
</tr>
</tbody>
</table>

### Event Data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addevent</code></td>
<td>Add event to <code>timeseries</code> object</td>
</tr>
<tr>
<td><code>delevent</code></td>
<td>Remove <code>tsdata.event</code> objects from <code>timeseries</code> object</td>
</tr>
<tr>
<td><code>gettsafteratevent</code></td>
<td>New <code>timeseries</code> object with samples occurring at or after event</td>
</tr>
<tr>
<td><code>gettsafterevent</code></td>
<td>New <code>timeseries</code> object with samples occurring after event</td>
</tr>
<tr>
<td><code>gettsatevent</code></td>
<td>New <code>timeseries</code> object with samples occurring at event</td>
</tr>
<tr>
<td><code>gettsbeforeatevent</code></td>
<td>New <code>timeseries</code> object with samples occurring before or at event</td>
</tr>
<tr>
<td><code>gettsbeforeevent</code></td>
<td>New <code>timeseries</code> object with samples occurring before event</td>
</tr>
<tr>
<td><code>gettsbetweenevents</code></td>
<td>New <code>timeseries</code> 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><code>iqr (timeseries)</code></td>
<td>Interquartile range of <code>timeseries</code> data</td>
</tr>
<tr>
<td><code>max (timeseries)</code></td>
<td>Maximum value of <code>timeseries</code> data</td>
</tr>
<tr>
<td><code>mean (timeseries)</code></td>
<td>Mean value of <code>timeseries</code> data</td>
</tr>
<tr>
<td><code>median (timeseries)</code></td>
<td>Median value of <code>timeseries</code> 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**

**General Purpose (p. 1-47)**
Query and set tscollection object properties, plot tscollection objects

**Data Manipulation (p. 1-48)**
Add or delete data, manipulate tscollection objects

**General Purpose**

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Types (p. 1-49)</td>
<td>Numeric, character, structures, cell arrays, and data type conversion</td>
</tr>
<tr>
<td>Data Type Conversion (p. 1-58)</td>
<td>Convert one numeric type to another, numeric to string, string to numeric, structure to cell array, etc.</td>
</tr>
<tr>
<td>Operators and Special Characters (p. 1-60)</td>
<td>Arithmetic, relational, and logical operators, and special characters</td>
</tr>
<tr>
<td>String Functions (p. 1-62)</td>
<td>Create, identify, manipulate, parse, evaluate, and compare strings</td>
</tr>
<tr>
<td>Bit-wise Functions (p. 1-65)</td>
<td>Perform set, shift, and, or, compare, etc. on specific bit fields</td>
</tr>
<tr>
<td>Logical Functions (p. 1-66)</td>
<td>Evaluate conditions, testing for true or false</td>
</tr>
<tr>
<td>Relational Functions (p. 1-66)</td>
<td>Compare values for equality, greater than, less than, etc.</td>
</tr>
<tr>
<td>Set Functions (p. 1-67)</td>
<td>Find set members, unions, intersections, etc.</td>
</tr>
<tr>
<td>Date and Time Functions (p. 1-67)</td>
<td>Obtain information about dates and times</td>
</tr>
<tr>
<td>Programming in MATLAB (p. 1-68)</td>
<td>M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling</td>
</tr>
</tbody>
</table>

### Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric Types (p. 1-50)</td>
<td>Integer and floating-point data</td>
</tr>
<tr>
<td>Characters and Strings (p. 1-51)</td>
<td>Characters and arrays of characters</td>
</tr>
<tr>
<td>Structures (p. 1-52)</td>
<td>Data of varying types and sizes stored in fields of a structure</td>
</tr>
</tbody>
</table>
Cell Arrays (p. 1-53) | Data of varying types and sizes stored in cells of array
Function Handles (p. 1-54) | Invoke a function indirectly via handle
MATLAB Classes and Objects (p. 1-55) | MATLAB object-oriented class system
Java Classes and Objects (p. 1-55) | Access Java classes through MATLAB interface
Data Type Identification (p. 1-57) | Determine data type of a variable

**Numeric Types**

arrayfun | Apply function to each element of array
cast | Cast variable to different data type
cat | Concatenate arrays along specified dimension
class | Create object or return class of object
find | Find indices and values of nonzero elements
intmax | Largest value of specified integer type
intmin | Smallest value of specified integer type
intwarning | Control state of integer warnings
ipermute | Inverse permute dimensions of N-D array
isa | Determine whether input is object of given class
isequal | Test arrays for equality
### Programming and Data Types

- `isequalwithnanans` Test arrays for equality, treating NaNs as equal
- `isfinite` Array elements that are finite
- `isinf` Array elements that are infinite
- `isnan` Array elements that are NaN
- `isnumeric` Determine whether input is numeric array
- `isreal` Determine whether input is real array
- `isscalar` Determine whether input is scalar
- `isvector` Determine whether input is vector
- `permute` Rearrange dimensions of N-D array
- `realmax` Largest positive floating-point number
- `realmin` Smallest positive floating-point number
- `reshape` Reshape array
- `squeeze` Remove singleton dimensions
- `zeros` Create array of all zeros

### Characters and Strings

See “String Functions” on page 1-62 for all string-related functions.

- `cellstr` Create cell array of strings from character array
- `char` Convert to character array (string)
- `eval` Execute string containing MATLAB expression
- `findstr` Find string within another, longer string
isstr
Determine whether input is character array

regexp, regexpi
Match regular expression

sprintf
Write formatted data to string

sscanf
Read formatted data from string

strcat
Concatenate strings horizontally

strcmp, strcmpi
Compare strings

strings
MATLAB 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
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
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>isequal</td>
<td>Test arrays for equality</td>
</tr>
<tr>
<td>iscalar</td>
<td>Determine whether input is scalar</td>
</tr>
<tr>
<td>isvector</td>
<td>Determine whether input is vector</td>
</tr>
<tr>
<td>mat2cell</td>
<td>Divide matrix into cell array of matrices</td>
</tr>
<tr>
<td>num2cell</td>
<td>Convert numeric array to cell array</td>
</tr>
<tr>
<td>struct2cell</td>
<td>Convert structure to cell array</td>
</tr>
</tbody>
</table>

**Function Handles**

<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>feval</td>
<td>Evaluate function</td>
</tr>
<tr>
<td>func2str</td>
<td>Construct function name string from function handle</td>
</tr>
<tr>
<td>functions</td>
<td>Information about function handle</td>
</tr>
<tr>
<td>function_handle (@)</td>
<td>Handle used in calling functions indirectly</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>str2func</td>
<td>Construct function handle from function name string</td>
</tr>
</tbody>
</table>
**MATLAB Classes and Objects**

- **class**: Create object or return class of object
- **fieldnames**: Field names of structure, or public fields of object
- **inferiorto**: Establish inferior class relationship
- **isa**: Determine whether input is object of given class
- **isobject**: Determine whether input is MATLAB OOPs object
- **loadobj**: User-defined extension of load function for user objects
- **methods**: Information on class methods
- **methodsviw**: Information on class methods in separate window
- **saveobj**: User-defined extension of save function for user objects
- **subsasgn**: Subscripted assignment for objects
- **subsindex**: Subscripted indexing for objects
- **subsref**: Subscripted reference for objects
- **substruct**: Create structure argument for subsasgn or subsref
- **superiorto**: Establish superior class relationship

**Java Classes and Objects**

- **cell**: Construct cell array
- **class**: Create object or return class of object
- **clear**: Remove items from workspace, freeing up system memory
- **depfun**: List dependencies of M-file or P-file
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exist</td>
<td>Check existence of variable, function, directory, or Java class</td>
</tr>
<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 Java import list</td>
</tr>
<tr>
<td>inmem</td>
<td>Names of M-files, MEX-files, 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 Java object</td>
</tr>
<tr>
<td>javaaddpath</td>
<td>Add entries to dynamic Java class path</td>
</tr>
<tr>
<td>javaArray</td>
<td>Construct Java array</td>
</tr>
<tr>
<td>javachk</td>
<td>Generate error message based on Java feature support</td>
</tr>
<tr>
<td>javaclasspath</td>
<td>Set and get dynamic Java class path</td>
</tr>
<tr>
<td>javaMethod</td>
<td>Invoke Java method</td>
</tr>
<tr>
<td>javaObject</td>
<td>Construct Java object</td>
</tr>
<tr>
<td>javarmpath</td>
<td>Remove entries from dynamic Java class path</td>
</tr>
<tr>
<td>methods</td>
<td>Information on class methods</td>
</tr>
<tr>
<td>methodsview</td>
<td>Information on class methods in separate window</td>
</tr>
<tr>
<td>usejava</td>
<td>Determine whether Java feature is supported in MATLAB</td>
</tr>
<tr>
<td>which</td>
<td>Locate functions and files</td>
</tr>
</tbody>
</table>
### 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 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 whether input is MATLAB OOPs object</td>
</tr>
<tr>
<td>isreal</td>
<td>Determine whether input is real array</td>
</tr>
<tr>
<td>isstr</td>
<td>Determine whether input is character array</td>
</tr>
<tr>
<td>isstruct</td>
<td>Determine whether input is structure array</td>
</tr>
<tr>
<td>who, whos</td>
<td>List variables in workspace</td>
</tr>
</tbody>
</table>
### Data Type Conversion

**Numeric (p. 1-58)**
- Convert data of one numeric type to another numeric type

**String to Numeric (p. 1-58)**
- Convert characters to numeric equivalent

**Numeric to String (p. 1-59)**
- Convert numeric to character equivalent

**Other Conversions (p. 1-59)**
- Convert to structure, cell array, function handle, etc.

### Numeric

- `cast`
  - Cast variable to different data type
- `double`
  - Convert to double precision
- `int8, int16, int32, int64`
  - Convert to signed integer
- `single`
  - Convert to single precision
- `typecast`
  - Convert data types without changing underlying data
- `uint8, uint16, uint32, uint64`
  - Convert to unsigned integer

### String to Numeric

- `base2dec`
  - Convert base N number string to decimal number
- `bin2dec`
  - Convert binary number string to decimal number
- `cast`
  - Cast variable to different data type
- `hex2dec`
  - Convert hexadecimal number string to decimal number
- `hex2num`
  - Convert hexadecimal number string to double-precision number
str2double  
Convert string to double-precision value

str2num  
Convert string to number

unicode2native  
Convert Unicode characters to numeric bytes

### Numeric to String

cast  
Cast variable to different data type

char  
Convert to character array (string)

dec2base  
Convert decimal to base N number in string

dec2bin  
Convert decimal to binary number in string

dec2hex  
Convert decimal to hexadecimal number in string

int2str  
Convert integer to string

mat2str  
Convert matrix to string

native2unicode  
Convert numeric bytes to Unicode characters

num2str  
Convert number to string

### 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
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>Convert numeric values to logical</td>
</tr>
<tr>
<td>mat2cell</td>
<td>Divide matrix into cell array of matrices</td>
</tr>
<tr>
<td>num2cell</td>
<td>Convert numeric array to cell array</td>
</tr>
<tr>
<td>num2hex</td>
<td>Convert singles and doubles to IEEE hexadecimal strings</td>
</tr>
<tr>
<td>str2func</td>
<td>Construct function handle from function name string</td>
</tr>
<tr>
<td>str2mat</td>
<td>Form blank-padded character matrix from strings</td>
</tr>
<tr>
<td>struct2cell</td>
<td>Convert structure to cell array</td>
</tr>
</tbody>
</table>

**Operators and Special Characters**

- **Arithmetic Operators (p. 1-60)**: Plus, minus, power, left and right divide, transpose, etc.
- **Relational Operators (p. 1-61)**: Equal to, greater than, less than or equal to, etc.
- **Logical Operators (p. 1-61)**: Element-wise and short circuit and, or, not
- **Special Characters (p. 1-62)**: Array constructors, line continuation, comments, etc.

**Arithmetic Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
</tr>
<tr>
<td>.</td>
<td>Decimal point</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
</tr>
<tr>
<td>*</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Matrix right division</td>
</tr>
</tbody>
</table>
\ 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
>= Greater than or equal to
== Equal to
~= Not equal to

**Logical Operators**

See also “Logical Functions” on page 1-66 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

**String Functions**

Description of Strings in MATLAB (p. 1-63) Basics of string handling in MATLAB

String Creation (p. 1-63) Create strings, cell arrays of strings, concatenate strings together

String Identification (p. 1-63) Identify characteristics of strings
| String Manipulation (p. 1-64) | Convert case, strip blanks, replace characters |
| String Parsing (p. 1-64) | Formatted read, regular expressions, locate substrings |
| String Evaluation (p. 1-65) | Evaluate stated expression in string |
| String Comparison (p. 1-65) | Compare contents of strings |

**Description of Strings in MATLAB**

```
strings MATLAB 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**

- **class**
  - Create object or return class of object
- **isa**
  - Determine whether input is object of given class
- **iscellstr**
  - Determine whether input is cell array of strings
- **ischar**
  - Determine whether item is character array
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**String Manipulation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debblank</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>
</tbody>
</table>
strread  Read formatted data from string
strtok  Selected parts of string

**String Evaluation**

eval  Execute string containing MATLAB expression
evalc  Evaluate MATLAB expression with capture
evalin  Execute MATLAB expression in specified workspace

**String Comparison**

strcmp, strcmpi  Compare strings
strmatch  Find possible matches for string
strncmp, strncmpi  Compare first n characters of strings

**Bit-wise Functions**

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 Functions

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

t
Logical 1 (true)

xor
Logical exclusive-OR

See “Operators and Special Characters” on page 1-60 for logical operators.

Relational Functions

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 1-60 for relational operators.

### Set Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersect</td>
<td>Find set intersection of two vectors</td>
</tr>
<tr>
<td>ismember</td>
<td>Array elements that are members of set</td>
</tr>
<tr>
<td>issorted</td>
<td>Determine whether set elements are in sorted order</td>
</tr>
<tr>
<td>setdiff</td>
<td>Find set difference of two vectors</td>
</tr>
<tr>
<td>setxor</td>
<td>Find set exclusive OR of two vectors</td>
</tr>
<tr>
<td>union</td>
<td>Find set union of two vectors</td>
</tr>
<tr>
<td>unique</td>
<td>Find unique elements of vector</td>
</tr>
</tbody>
</table>

### Date and Time Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addtodate</td>
<td>Modify date number by field</td>
</tr>
<tr>
<td>calendar</td>
<td>Calendar for specified month</td>
</tr>
<tr>
<td>clock</td>
<td>Current time as date vector</td>
</tr>
<tr>
<td>cputime</td>
<td>Elapsed CPU time</td>
</tr>
<tr>
<td>date</td>
<td>Current date string</td>
</tr>
<tr>
<td>datenum</td>
<td>Convert date and time to serial date number</td>
</tr>
<tr>
<td>datestr</td>
<td>Convert date and time to string format</td>
</tr>
<tr>
<td>datevec</td>
<td>Convert date and time to vector of components</td>
</tr>
</tbody>
</table>
eomday
etime
now
weekday

Last day of month
Time elapsed between date vectors
Current date and time
Day of week

Programming in MATLAB

M-File Functions and Scripts
(p. 1-68)

Declare functions, handle arguments, identify dependencies, etc.

Evaluation of Expressions and Functions (p. 1-70)

Evaluate expression in string, apply function to array, run script file, etc.

Timer Functions (p. 1-71)

Schedule execution of MATLAB commands

Variables and Functions in Memory
(p. 1-71)

List files in memory, clear M-files in memory, assign to variable in nondefault workspace, refresh caches

Control Flow (p. 1-72)

if-then-else, for loops, switch-case, try-catch

Error Handling (p. 1-73)

Generate warnings and errors, test for and catch errors, retrieve most recent error message

MEX Programming (p. 1-74)

Compile MEX function from C or Fortran code, list MEX-files in memory, debug MEX-files

M-File Functions and Scripts

addOptional (inputParser)

Add optional argument to inputParser schema

addParamValue (inputParser)

Add parameter-value argument to inputParser schema
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>mfilename</td>
<td>Name of currently running M-file</td>
</tr>
<tr>
<td>namelengthmax</td>
<td>Maximum identifier length</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 preparsed pseudocode 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 of Expressions and Functions

<table>
<thead>
<tr>
<th>Function</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>
<tr>
<td>evalin</td>
<td>Execute MATLAB expression in specified workspace</td>
</tr>
<tr>
<td>feval</td>
<td>Evaluate function</td>
</tr>
<tr>
<td>iskeyword</td>
<td>Determine whether input is MATLAB keyword</td>
</tr>
<tr>
<td>isvarname</td>
<td>Determine whether input is valid variable name</td>
</tr>
<tr>
<td>pause</td>
<td>Halt execution temporarily</td>
</tr>
<tr>
<td>run</td>
<td>Run script that is not on current path</td>
</tr>
<tr>
<td>script</td>
<td>Script M-file description</td>
</tr>
<tr>
<td>structfun</td>
<td>Apply function to each field of scalar structure</td>
</tr>
</tbody>
</table>
symvar
Determine symbolic variables in expression

tic, toc
Measure performance using stopwatch timer

**Timer Functions**
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)
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, Java classes in memory

isglobal

Determine whether input is global variable

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

persistent

Define persistent variable

rehash

Refresh function and file system path caches

Control Flow

break

Terminate execution of for or while loop

case

Execute block of code if condition is true

catch

Specify how to respond to error in try statement

continue

Pass control to next iteration of for or while loop

else

Execute statements if condition is false
elseif
end
error
for
if
otherwise
return
switch
try
while

**Error Handling**

assert
catch
error
ferror
intwarning
lasterr
lasterror
lastwarn
rethrow
try
warning

Last warning message
Reissue error
Attempt to execute block of code, and catch errors
Warning message

MEX Programming

dbmex
inmem
mex
mexext

Enable MEX-file debugging
Names of M-files, MEX-files, Java classes in memory
Compile MEX-function from C or Fortran source code
MEX-filename extension
File I/O

File Name Construction (p. 1-75)  Get path, directory, filename information; construct filenames
Opening, Loading, Saving Files (p. 1-76)  Open files; transfer data between files and MATLAB workspace
Memory Mapping (p. 1-76)  Access file data via memory map using MATLAB array indexing
Low-Level File I/O (p. 1-76)  Low-level operations that use a file identifier
Text Files (p. 1-77)  Delimited or formatted I/O to text files
XML Documents (p. 1-78)  Documents written in Extensible Markup Language
Spreadsheets (p. 1-78)  Excel and Lotus 1-2-3 files
Scientific Data (p. 1-79)  CDF, FITS, HDF formats
Audio and Audio/Video (p. 1-80)  General audio functions; SparcStation, WAVE, AVI files
Images (p. 1-82)  Graphics files
Internet Exchange (p. 1-83)  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

filemarker  Character to separate file name and internal function name
fileparts  Parts of file name and path
filesep  Directory separator for current platform
fullfile  Build full filename from parts
Functions — By Category

```
tempdir
Name of system’s temporary directory

tempname
Unique name for temporary file

Opening, Loading, Saving Files

daqread
Read Data Acquisition Toolbox (.daq) file

filehandle
Construct file handle object

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 about errors in file input or output
```
fgetl Read line from file, discarding newline character
fgets Read line from file, keeping newline character
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 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
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 Functions (p. 1-78)
Read and write Microsoft Excel spreadsheet

Lotus 1-2-3 Functions (p. 1-78)
Read and write Lotus WK1 spreadsheet

**Microsoft Excel Functions**

xlsfinfo
Determine whether file contains Microsoft Excel (.xls) spreadsheet

xlsread
Read Microsoft Excel spreadsheet file (.xls)

xlswrite
Write Microsoft Excel spreadsheet file (.xls)

**Lotus 1-2-3 Functions**

wk1finfo
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

- **Common Data Format (CDF)** *(p. 1-79)*
  - Work with CDF files
- **Flexible Image Transport System** *(p. 1-79)*
  - Work with FITS files
- **Hierarchical Data Format (HDF)** *(p. 1-80)*
  - Work with HDF files
- **Band-Interleaved Data** *(p. 1-80)*
  - Work with band-interleaved files

### Common Data Format (CDF)

- `cdfepoch`
  - Construct `cdfepoch` object for Common Data Format (CDF) export
- `cdfinfo`
  - Information about Common Data Format (CDF) file
- `cdfread`
  - Read data from Common Data Format (CDF) file
- `cdfwrite`
  - Write data to Common Data Format (CDF) file
- `todatenum`
  - Convert CDF epoch object to MATLAB datenum

### Flexible Image Transport System

- `fitsinfo`
  - Information about FITS file
- `fitsread`
  - Read data from FITS file
Hierarchical Data Format (HDF)

- **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

- **General (p. 1-81)**: Create audio player object, obtain information about multimedia files, convert to/from audio signal
- **SPARCstation-Specific Sound Functions (p. 1-81)**: Access NeXT/SUN (.au) sound files
Microsoft WAVE Sound Functions (p. 1-81)
Access Microsoft WAVE (.wav) sound files

Audio/Video Interleaved (AVI) Functions (p. 1-82)
Access Audio/Video interleaved (.avi) sound files

### General

- **audioplayer**
  - Create audio player object
- **audiorecorder**
  - Create audio recorder object
- **beep**
  - Produce beep sound
- **lin2mu**
  - Convert linear audio signal to mu-law
- **mmfileinfo**
  - Information about multimedia file
- **mu2lin**
  - Convert mu-law audio signal to linear
- **sound**
  - Convert vector into sound
- **soundsc**
  - Scale data and play as sound

### SPARCstation-Specific Sound Functions

- **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 Functions

- **wavfinfo**
  - Information about Microsoft WAVE (.wav) sound file
- **wavplay**
  - Play recorded sound on PC-based audio output device
Functions — By Category

wavread  Read Microsoft WAVE (.wav) sound file
wavrecord Record sound using PC-based audio input device
wavwrite Write Microsoft WAVE (.wav) sound file

Audio/Video Interleaved (AVI) Functions

addframe  Add frame to Audio/Video Interleaved (AVI) file
avifile Create new Audio/Video Interleaved (AVI) file
aviinfo Information about Audio/Video Interleaved (AVI) file
aviread Read Audio/Video Interleaved (AVI) file
close (avifile) Close Audio/Video Interleaved (AVI) file
movie2avi Create Audio/Video Interleaved (AVI) movie from MATLAB movie

Images

exifread  Read EXIF information from JPEG and TIFF image files
im2java Convert image to Java image
imfinfo Information about graphics file
imread Read image from graphics file
imwrite Write image to graphics file
Internet Exchange

URL, Zip, Tar, E-Mail (p. 1-83)
Send e-mail, read from given URL, extract from tar or zip file, compress and decompress files

FTP Functions (p. 1-83)
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 Functions

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
ftp  Connect to FTP server, creating FTP object
mget  Download file from FTP server
mkdir (ftp)  Create new directory on FTP server
mput  Upload file or directory to FTP server
rename  Rename file on FTP server
rmdir (ftp)  Remove directory on FTP server
## Graphics

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Plots and Graphs (p. 1-85)</td>
<td>Linear line plots, log and semilog plots</td>
</tr>
<tr>
<td>Plotting Tools (p. 1-86)</td>
<td>GUIs for interacting with plots</td>
</tr>
<tr>
<td>Annotating Plots (p. 1-86)</td>
<td>Functions for and properties of titles, axes labels, legends, mathematical symbols</td>
</tr>
<tr>
<td>Specialized Plotting (p. 1-87)</td>
<td>Bar graphs, histograms, pie charts, contour plots, function plotters</td>
</tr>
<tr>
<td>Bit-Mapped Images (p. 1-91)</td>
<td>Display image object, read and write graphics file, convert to movie frames</td>
</tr>
<tr>
<td>Printing (p. 1-91)</td>
<td>Printing and exporting figures to standard formats</td>
</tr>
<tr>
<td>Handle Graphics (p. 1-92)</td>
<td>Creating graphics objects, setting properties, finding handles</td>
</tr>
</tbody>
</table>

### Basic Plots and Graphs

- `box` : Axes border
- `errorbar` : Plot error bars along curve
- `hold` : Retain current graph in figure
- `LineSpec` : Line specification string syntax
- `loglog` : Log-log scale plot
- `plot` : 2-D line plot
- `plot3` : 3-D line plot
- `plotyy` : 2-D line plots with y-axes on both left and right side
- `polar` : Polar coordinate plot
semilogx, semilogy  
Semilogarithmic plots

subplot  
Create axes in tiled positions

Plotting Tools

figurepalette  
Show or hide figure palette

pan  
Pan view of graph interactively

plotbrowser  
Show or hide figure plot browser

plotedit  
Interactively edit and annotate plots

plottools  
Show or hide plot tools

propertyeditor  
Show or hide property editor

rotate3d  
Rotate 3-D view using mouse

showplottool  
Show or hide figure plot tool

zoom  
Turn zooming on or off or magnify by factor

Annotating Plots

annotation  
Create annotation objects

clabel  
Contour plot elevation labels

datacursormode  
Enable or disable interactive data cursor mode

datetick  
Date formatted tick labels

gtext  
Mouse placement of text in 2-D view

legend  
Graph legend for lines and patches

line  
Create line object

rectangle  
Create 2-D rectangle object

texlabel  
Produce TeX format from character string
title
xlabel, ylabel, zlabel

Add title to current axes
Label x-, y-, and z-axis

**Specialized Plotting**

Area, Bar, and Pie Plots (p. 1-87)
Contour Plots (p. 1-88)
Direction and Velocity Plots (p. 1-88)
Discrete Data Plots (p. 1-88)
Function Plots (p. 1-88)
Histograms (p. 1-89)
Polygons and Surfaces (p. 1-89)
Scatter/Bubble Plots (p. 1-90)
Animation (p. 1-90)

1-D, 2-D, and 3-D graphs and charts
Unfilled and filled contours in 2-D and 3-D
Comet, compass, feather and quiver plots
Stair, step, and stem plots
Easy-to-use plotting utilities for graphing functions
Plots for showing distributions of data
Functions to generate and plot surface patches in two or more dimensions
Plots of point distributions
Functions to create and play movies of plots

**Area, Bar, and Pie Plots**

area
bar, barh
bar3, bar3h
pareto
pie
pie3

Filled area 2-D plot
Plot bar graph (vertical and horizontal)
Plot 3-D bar chart
Pareto chart
Pie chart
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
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 | Filled 2-D polygons
fill3 | Filled 3-D polygons
inpolygon | Points inside polygonal region
pcolor | Pseudocolor (checkerboard) plot
polyarea | Area of polygon
rectint | Rectangle intersection area
ribbon | Ribbon plot
slice | Volumetric slice plot
sphere | Generate sphere
tsearch | Search for enclosing Delaunay triangle
tsearchn | N-D closest simplex search
voronoi | Voronoi diagram
waterfall | Waterfall plot

**Scatter/Bubble Plots**

plotmatrix | Scatter plot matrix
scatter | Scatter plot
scatter3 | 3-D scatter plot

**Animation**

frame2im | Convert movie frame to indexed image
getframe | Capture movie frame
im2frame | Convert image to movie frame
movie
noanimate

Bit-Mapped Images

frame2im
im2frame
im2java
image
imagesc
imfinfo
imformats
imread
imwrite
ind2rgb

Printing

frameedit
hgexport
orient
print, printopt
printdlg

Play recorded movie frames
Change EraseMode of all objects to normal

Convert movie frame to indexed image
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

Edit print frames for Simulink and Stateflow block diagrams
Export figure
Hardcopy paper orientation
Print figure or save to file and configure printer defaults
Print dialog box
printpreview
saveas

**Handle Graphics**

Finding and Identifying Graphics Objects (p. 1-92)
Object Creation Functions (p. 1-93)
Plot Objects (p. 1-93)
Figure Windows (p. 1-94)
Axes Operations (p. 1-95)
Operating on Object Properties (p. 1-95)

**Finding and Identifying Graphics Objects**

allchild
ancestor
copyobj
delete
findall
findfigs
findobj
gca
gcbf

Find all children of specified objects
Ancestor of graphics object
Copy graphics objects and their descendants
Remove files or graphics objects
Find all graphics objects
Find visible offscreen figures
Locate graphics objects with specific properties
Current axes handle
Handle of figure containing object whose callback is executing
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcbo</td>
<td>Handle of object whose callback is executing</td>
</tr>
<tr>
<td>gco</td>
<td>Handle of current object</td>
</tr>
<tr>
<td>get</td>
<td>Query object properties</td>
</tr>
<tr>
<td>ishandle</td>
<td>Is object handle valid</td>
</tr>
<tr>
<td>propedit</td>
<td>Open Property Editor</td>
</tr>
<tr>
<td>set</td>
<td>Set object properties</td>
</tr>
</tbody>
</table>

**Object Creation Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>Create axes graphics object</td>
</tr>
<tr>
<td>figure</td>
<td>Create figure graphics object</td>
</tr>
<tr>
<td>hggroup</td>
<td>Create hggroup object</td>
</tr>
<tr>
<td>hgtransform</td>
<td>Create hgtransform graphics object</td>
</tr>
<tr>
<td>image</td>
<td>Display image object</td>
</tr>
<tr>
<td>light</td>
<td>Create light object</td>
</tr>
<tr>
<td>line</td>
<td>Create line object</td>
</tr>
<tr>
<td>patch</td>
<td>Create patch graphics object</td>
</tr>
<tr>
<td>rectangle</td>
<td>Create 2-D rectangle object</td>
</tr>
<tr>
<td>root object</td>
<td>Root object properties</td>
</tr>
<tr>
<td>surface</td>
<td>Create surface object</td>
</tr>
<tr>
<td>text</td>
<td>Create text object in current axes</td>
</tr>
<tr>
<td>uicontextmenu</td>
<td>Create context menu</td>
</tr>
</tbody>
</table>

**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   |
| Annotation Rectangle Properties| Define annotation rectangle properties |
| Annotation Textarrow Properties| Define annotation textarrow properties |
| Annotation Textbox Properties | Define annotation textbox properties |
| Areaseries Properties         | Define areaseries properties        |
| Barseries Properties          | Define barseries properties         |
| Contourgroup Properties       | Define contourgroup properties       |
| Errorbarseries Properties     | Define errorbarseries properties     |
| Image Properties              | Define image properties              |
| Lineseries Properties         | Define lineseries properties         |
| Quivergroup Properties        | Define quivergroup properties        |
| Scattergroup Properties       | Define scattergroup properties       |
| Stairseries Properties        | Define stairseries properties        |
| Stemseries Properties         | Define stemseries properties         |
| Surfaceplot Properties        | Define surfaceplot properties        |

**Figure Windows**

- `clf` Clear current figure window
- `close` Remove specified figure
- `closereq` Default figure close request function
- `drawnow` Complete pending drawing events
- `gcf` Current figure handle
- `hgload` Load Handle Graphics object hierarchy from file
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hgsave</td>
<td>Save Handle Graphics object hierarchy to file</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>Function</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>

**Operating on Object Properties**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>Query 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 object properties</td>
</tr>
</tbody>
</table>
3-D Visualization

Surface and Mesh Plots (p. 1-96)  
Plot matrices, visualize functions of two variables, specify colormap

View Control (p. 1-98)  
Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits

Lighting (p. 1-100)  
Add and control scene lighting

Transparency (p. 1-100)  
Specify and control object transparency

Volume Visualization (p. 1-101)  
Visualize gridded volume data

Surface and Mesh Plots

Creating Surfaces and Meshes (p. 1-96)  
Visualizing gridded and triangulated data as lines and surfaces

Domain Generation (p. 1-97)  
Gridding data and creating arrays

Color Operations (p. 1-97)  
Specifying, converting, and manipulating color spaces, colormaps, colorbars, and backgrounds

Colormaps (p. 1-98)  
Built-in colormaps you can use

Creating Surfaces and Meshes

hidden  
Remove hidden lines from mesh plot

mesh, meshc, meshz  
Mesh plots

peaks  
Example function of two variables

surf, surfc  
3-D shaded surface plot

surface  
Create surface object

surfl  
Surface plot with colormap-based lighting
tetramesh
trimesh
triplot
trisurf

Domain Generation
griddata
meshgrid

Color Operations
brighten
caxis
colorbar
colordef
colormap
colormapeditor
ColorSpec
graymon
hsv2rgb
rgb2hsv
rgbplot
shading
spinmap

Tetrahedron mesh plot
Triangular mesh plot
2-D triangular plot
Triangular surface plot
Data gridding
Generate X and Y arrays for 3-D plots
Brighten or darken colormap
Color axis scaling
Colorbar showing color scale
Set default property values to display different color schemes
Set and get current colormap
Start colormap editor
Color specification
Set default figure properties for grayscale monitors
Convert HSV colormap to RGB colormap
Convert RGB colormap to HSV colormap
Plot colormap
Set color shading properties
Spin colormap
surfnorm  Compute and display 3-D surface normals
whitebg   Change axes background color

**Colormaps**

contrast  Grayscale colormap for contrast enhancement

**View Control**

Controlling the Camera Viewpoint (p. 1-98)  Orbiting, dollying, pointing, rotating camera positions and setting fields of view
Setting the Aspect Ratio and Axis Limits (p. 1-99)  Specifying what portions of axes to view and how to scale them
Object Manipulation (p. 1-99)  Panning, rotating, and zooming views
Selecting Region of Interest (p. 1-100)  Interactively identifying rectangular regions

**Controlling the 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
camproj
camroll
camtarget
camup
camva
camzoom
makehgtform
view
viewmtx

campos
Set or query camera position
camproj
Set or query projection type
camroll
Rotate camera about view axis
camtarget
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
makehgtform
Create 4-by-4 transform matrix
view
Viewpoint specification
viewmtx
View transformation matrices

Setting the 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
Selecting Region of Interest

- dragrect: Drag rectangles with mouse
- rbbox: Create rubberband box for area selection

Lighting

- camlight: Create or move light object in camera coordinates
- diffuse: Calculate diffuse reflectance
- light: Create light object
- lightangle: Create or position light object in spherical coordinates
- lighting: Specify lighting algorithm
- material: Control reflectance properties of surfaces and patches
- specular: Calculate specular reflectance

Transparency

- alim: Set or query axes alpha limits
- alpha: Set transparency properties for objects in current axes
- alphamap: Specify figure alphamap (transparency)
## 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>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 the 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>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------</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>
Creating Graphical User Interfaces

Predefined Dialog Boxes (p. 1-103)
Dialogue boxes for error, user input, waiting, etc.

Deploying User Interfaces (p. 1-104)
Launch GUIs, create the handles structure

Developing User Interfaces (p. 1-104)
Start GUIDE, manage application data, get user input

User Interface Objects (p. 1-105)
Create GUI components

Finding Objects from Callbacks (p. 1-106)
Find object handles from within callbacks functions

GUI Utility Functions (p. 1-106)
Move objects, wrap text

Controlling Program Execution (p. 1-107)
Wait and resume based on user input

Predefined Dialog Boxes

dialog
Create and display dialog box

erordlg
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 a directory
Functions — By Category

**uigetfile**
Open standard dialog box for retrieving files

**uigetpref**
Open dialog box for retrieving preferences

**uiopen**
Open file selection dialog box with appropriate file filters

**uiputfile**
Open standard dialog box for saving files

**uisave**
Open standard dialog box for saving workspace variables

**uisetcolor**
Open standard dialog box for setting object's ColorSpec

**uisetfont**
Open standard dialog box for setting object's font characteristics

**waitbar**
Open waitbar

**warndlg**
Open warning dialog box

---

**Deploying User Interfaces**

**guidata**
Store or retrieve GUI data

**guihandles**
Create structure of handles

**movegui**
Move GUI figure to specified location on screen

**openfig**
Open new copy or raise existing copy of saved figure

---

**Developing User Interfaces**

**addpref**
Add preference

**getappdata**
Value of application-defined data

**getpref**
Preference
Creating Graphical User Interfaces

**User Interface Objects**

- `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

- `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
Functions — By Category

- **uimenu**  
  Create menus on figure windows

- **uipanel**  
  Create panel container object

- **uipushtool**  
  Create push button on toolbar

- **uitoggletool**  
  Create toggle button on toolbar

- **uitoobar**  
  Create toolbar on figure

**Finding Objects from Callbacks**

- **findall**  
  Find all graphics objects

- **findfigs**  
  Find visible offscreen figures

- **findobj**  
  Locate graphics objects with specific properties

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

- **gcbo**  
  Handle of object whose callback is executing

**GUI Utility Functions**

- **align**  
  Align user interface controls (uicontrols) and axes

- **getpixelposition**  
  Get component position in pixels

- **listfonts**  
  List available system fonts

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

- **setpixelposition**  
  Set component position in pixels

- **textwrap**  
  Wrapped string matrix for given uicontrol

- **uistack**  
  Reorder visual stacking order of objects
Controlling Program Execution

uiresume, uiwait  Control program execution
External Interfaces

Dynamic Link Libraries (p. 1-108)
Access functions stored in external shared library (.dll) files

Java (p. 1-109)
Work with objects constructed from Java API and third-party class packages

Component Object Model and ActiveX (p. 1-110)
Integrate COM components into your application

Dynamic Data Exchange (p. 1-112)
Communicate between applications by establishing a DDE conversation

Web Services (p. 1-113)
Communicate between applications over a network using SOAP and WSDL

Serial Port Devices (p. 1-113)
Read and write to devices connected to your computer's serial port

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

Dynamic Link Libraries

calllib
Call function in external library

libfunctions
Information on functions in external library

libfunctionsview
Create window displaying information on functions in external library

libisloaded
Determine whether external library is loaded

libpointer
Create pointer object for use with external libraries
### External Interfaces

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libstruct</td>
<td>Construct structure as defined in external library</td>
</tr>
<tr>
<td>loadlibrary</td>
<td>Load external library into MATLAB</td>
</tr>
<tr>
<td>unloadlibrary</td>
<td>Unload external library from memory</td>
</tr>
</tbody>
</table>

### Java

<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>fieldnames</td>
<td>Field names of structure, or public fields of object</td>
</tr>
<tr>
<td>import</td>
<td>Add package or class to current Java import list</td>
</tr>
<tr>
<td>inspect</td>
<td>Open Property Inspector</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 Java object</td>
</tr>
<tr>
<td>ismethod</td>
<td>Determine whether input is object method</td>
</tr>
<tr>
<td>isprop</td>
<td>Determine whether input is object property</td>
</tr>
<tr>
<td>javaaddpath</td>
<td>Add entries to dynamic Java class path</td>
</tr>
<tr>
<td>javaArray</td>
<td>Construct Java array</td>
</tr>
<tr>
<td>javachk</td>
<td>Generate error message based on Java feature support</td>
</tr>
<tr>
<td>javaclasspath</td>
<td>Set and get dynamic Java class path</td>
</tr>
<tr>
<td>javaMethod</td>
<td>Invoke Java method</td>
</tr>
<tr>
<td>javaObject</td>
<td>Construct Java object</td>
</tr>
</tbody>
</table>
**javarmpath**
Remove entries from dynamic Java class path

**methods**
Information on class methods

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

**usejava**
Determine whether Java feature is supported in MATLAB

---

**Component Object Model and ActiveX**

**actxcontrol**
Create ActiveX control in figure window

**actxcontrollist**
List all currently installed ActiveX controls

**actxcontrolselect**
Open GUI to create ActiveX control

**actxGetRunningServer**
Get handle to running instance of Automation server

**actxserver**
Create COM server

**addproperty**
Add custom property to object

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

**delete (COM)**
Remove COM control or server

**deleteproperty**
Remove custom property from object

**enableservice**
Enable, disable, or report status of Automation server; enable DDE server

**eventlisteners**
List of events attached to listeners

**events**
List of events control can trigger

**Execute**
Execute MATLAB command in server

**Feval (COM)**
Evaluate MATLAB function in server
fieldnames
get (COM)
GetCharArray
GetFullMatrix
GetVariable
GetWorkspaceData
inspect
interfaces
invoke
isa
iscom
isevent
isinterface
ismethod
isprop
load (COM)
MaximizeCommandWindow
methods
methodsvie
MinimizeCommandWindow

Field names of structure, or public fields of object
Get property value from interface, or display properties
Get character array from server
Get matrix from server
Get data from variable in server workspace
Get data from server workspace
Open Property Inspector
List custom interfaces to COM server
Invoke method on object or interface, or display methods
Determine whether input is object of given class
Is input COM object
Is input event
Is input COM interface
Determine whether input is object method
Determine whether input is object property
Initialize control object from file
Open server window on Windows desktop
Information on class methods
Information on class methods in separate window
Minimize size of server window
move Move or resize control in parent window

propedit (COM) Open built-in property page for control

PutCharArray Store character array in server

PutFullMatrix Store matrix in server

PutWorkspaceData Store data in server workspace

Quit (COM) Terminate MATLAB server

registerevent Register event handler with control’s event

release Release interface

save (COM) Serialize control object to file

send Return list of events control can trigger

set (COM) Set object or interface property to specified value

unregisterallevents Unregister all events for control

unregisterevent Unregister event handler with control’s event

**Dynamic Data Exchange**

ddeadv Set up advisory link

ddeexec Send string for execution

ddeinit Initiate Dynamic Data Exchange (DDE) conversation

ddepoke Send data to application

ddereq Request data from application
**External Interfaces**

ddeterm  
Terminate Dynamic Data Exchange (DDE) conversation

ddeunadv  
Release advisory link

**Web Services**

callSoapService  
Send SOAP message off to endpoint

cREATECLASSFROMWSDL  
Create MATLAB object based on WSDL file

cREATESOAPMESSAGE  
Create SOAP message to send to server

parsesoapresponse  
Convert response string from SOAP server into MATLAB data types

**Serial Port Devices**

clear (serial)  
Remove serial port object from MATLAB workspace

delete (serial)  
Remove serial port object from memory

disp (serial)  
Serial port object summary information

fCLOSE (serial)  
Disconnect serial port object from device

fGETL (serial)  
Read line of text from device and discard terminator

fGETS (serial)  
Read line of text from device and include terminator

fOPEN (serial)  
Connect serial port object to device

fprintf (serial)  
Write text to device

fREAD (serial)  
Read binary data from device
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fscanf (serial)</td>
<td>Read data from device, and format as text</td>
</tr>
<tr>
<td>fwrite (serial)</td>
<td>Write binary data to device</td>
</tr>
<tr>
<td>get (serial)</td>
<td>Serial port object properties</td>
</tr>
<tr>
<td>instrcallback</td>
<td>Event information when event occurs</td>
</tr>
<tr>
<td>instrfind</td>
<td>Read serial port objects from memory to MATLAB workspace</td>
</tr>
<tr>
<td>instrfindall</td>
<td>Find visible and hidden serial port objects</td>
</tr>
<tr>
<td>isvalid (serial)</td>
<td>Determine whether serial port objects are valid</td>
</tr>
<tr>
<td>length (serial)</td>
<td>Length of serial port object array</td>
</tr>
<tr>
<td>load (serial)</td>
<td>Load serial port objects and variables into MATLAB workspace</td>
</tr>
<tr>
<td>readasync</td>
<td>Read data asynchronously from device</td>
</tr>
<tr>
<td>record</td>
<td>Record data and event information to file</td>
</tr>
<tr>
<td>save (serial)</td>
<td>Save serial port objects and variables to MAT-file</td>
</tr>
<tr>
<td>serial</td>
<td>Create serial port object</td>
</tr>
<tr>
<td>serialbreak</td>
<td>Send break to device connected to serial port</td>
</tr>
<tr>
<td>set (serial)</td>
<td>Configure or display serial port object properties</td>
</tr>
<tr>
<td>size (serial)</td>
<td>Size of serial port object array</td>
</tr>
<tr>
<td>stopasync</td>
<td>Stop asynchronous read and write operations</td>
</tr>
</tbody>
</table>
Functions — 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
addevent
addframe
addOptional (inputParser)
addParamValue (inputParser)
addpath
addpref
addproperty
addRequired (inputParser)
addsample
addsampletocollection
adddate
date
dt
airy
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
audioplayer
audiorecorder
aufinfo
auread
auwrite
avifile
aviinfo
aviread
axes
Axes Properties
axis
balance
bar, barh
bar3, bar3h
Barseries Properties
base2dec
beep
besselh
besseli
besselj
besselk
bessely
beta
betainc
betaln
bicg
bicgstab
bin2dec
binary
bitand
bitcmp
bitget
bitmax
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
builddocsearchdb
builtin
bsxfun
bvp4c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit
campan
campos
camproj
camroll
camtarget
camup
camva
camzoom
cart2pol
cart2sph
case
cast
cat
catch
caxis
cd
cd (ftp)
cdf2rdf
cdfepoch
cdfinfo
cdfread
cdfwrite
cceil
cell
cell2mat
cell2struct
celldisp
cellfun
cellplot
cellstr
cgstr
char
checkin
checkout
chol
cholinc
cholupdate
circshift
cla
clabel
class
clc
clear
clear (serial)
clf
clipboard
clock
close
close (avifile)
close (ftp)
closereq
cmopts
colamd
colmmd
colorbar
colordef
colormap
colormapeditor
ColorSpec
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
corrcoef
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
createClassFromWsdl
createCopy (inputParser)
createSoapMessage
cross
csc
cscd
csch
csvread
csvwrite
ctranspose (timeseries)
cumprod
cumsum
cumtrapz
curl
customverctrl
cylinder
daqread
daspect
datacursorsmode
datatipinfo
date
datenum
datestr
datetick
datevec
dbclear
dbcont
dbdown
dblquad
dbmex
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeadv
ddeexec
ddeget
ddeinit
ddepoke
ddereq
ddesd
ddeset
ddetermin
ddetermin
ddeunadv
deal
deblank
debug
dec2base
dec2bin
dec2hex
decic
decconv
del2
delaunay
delaunay3
delaunayn
delete
delete (COM)
delete (ftp)
delete (serial)
delete (timer)
deleteproperty
delevent
delsample
delsamplefromcollection
demo
depdir
depfun
det
detrend
detrend (timeseries)
deval
diag
dialog
diary
diff
diffuse
dir
dir (ftp)
disp
disp (serial)
disp (timer)
display
divergence
dlmread
dlmwrite
dmperm
doc
docopt
docsearch
dos
dot
double
dragrect
drawnow
dsearch
dsearchn
echo
echodemo
edit
eig
eigs
ellipj
ellipke
ellipsoid
else
elseif
enableservice
end
eomday
eps
eq
erf, erfc, erfcx, erfinv, erfcinv
error
errorbar
Errorbarseries Properties
errordlg
etime
etree
etreeplot
eval
evalc
evalin
eventlisteners
events
Execute
exifread
exist
exit
exp
expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh
ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
factor
factorial
false
fclose
fclose (serial)
fclose (serial)
feather
feof
ferror
feval
Feval (COM)
fft
fft2
fftn
fftwshift
fftw
fgetl
fgetl (serial)
fgets
fgets (serial)
fieldnames
figure
Figure Properties
figurepalette
fileattrib
filebrowser
File Formats
filemarker
fileparts
filehandle
filesep
fill
fill3
filter
filter (timeseries)
filter2
find
findall
findfigs
findobj
findstr
finish
fitsinfo
fitsread
fix
flipdim
fliplr
flipud
floor
flops
flow
fminbnd
fminsearch
fopen
fopen (serial)
for
format
fplot
fprintf
get (timeseries)
get (tscollection)
getabstime (timeseries)
getabstime (tscollection)
getappdata
GetCharArray
getdatasamplesize
getenv
getfield
getframe
GetFullMatrix
getinterpmethod
getpixelposition
getpref
getqualitydesc
getsampleusingtime (timeseries)
getsampleusingtime (tscollection)
gettimeseriesnames
gettafteratevent
gettafterevent
gettsatevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents
GetVariable
GetWorkspaceData
ginput
global
gmres
gplot
grabcode
gradient
graymon
grid
griddata
griddata3
griddatan
gsvd
gt
gtext
guidata
guide
guihandles
unzip
gzip
hadamard
hankel
hdf
hdf5
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
hex2dec
hex2num
hgexport
hggroup
Hggroup Properties
hgload
hgsave
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
imfinfo
imformats
import
importdata
imread
imwrite
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*
isa
isappdata
iscell
iscellstr
ischar
iscom
isdir
isempty
isempty (timeseries)
isempty (tscollection)
isequal
isequalwithnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishold
isinf
isinteger
isinterface
isjava
iskeyword
isletter
islogical
ismac
ismember
ismethod
isnan
isnumeric
isobject
isocaps
isocolors
isonormals
isosurface
ispc
ispref
isprime
isprop
isreal
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
isunix
isvalid (serial)
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
javaMethod
javaObject
javarmpath
keyboard
kron
lasterr
lasterror
lastwarn
lcm
ldl
ldivide, rdivide
le
legend
legendre
length
length (serial)
length (timeseries)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec
linkaxes
linkprop
linsolve
linspace
listedlg
listfonts
load
load (COM)
load (serial)
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor
lower
ls
lscov
lsqnonneg
lsqr
lt
lu
luinc
magic
makehgtform
mat2cell
mat2str
material
matlabcolon (matlab:)
matlabrc
matlabroot
matlab (UNIX)
matlab (Windows)
max
max (timeseries)
MaximizeCommandWindow
mean
mean (timeseries)
median
median (timeseries)
disp (memmapfile)
get (memmapfile)
memmapfile
memory
menu
mesh, meshc, meshz
meshgrid
methods
methodview
mex
mexext
mfilename
mget
min
min (timeseries)
MinimizeCommandWindow
minres
mislocked
mkdir
mkdir
mkdir (ftp)
mkpp
mldivide \\, mrdivide /  
mlint  
mlintrpt  
mlock  
mmfileinfo  
mod  
mode  
more  
mov e  
movefile  
movegui  
movie  
movie2avi  
mput  
msgbox  
mtimes  
mu2lin  
multibandread  
multibandwrite  
munlock  
namelengthmax  
NaN  
nargchk  
nargin, nargout  
nargoutchk  
native2unicode  
nchoosek  
ngrid  
ndims  
ne  
newplot  
extpow2  
nnz  
nonanimate  
nonzeros  
norm  
normest
partialpath
pascal
patch
Patch Properties
path
path2rc
pathdef
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
pol2cart
polar
dy
polyarea
dyder
dyeyeig
dyfit
dyint
dyval
dyvalm
dyw2
dywer
dpow2
ppval
prefdir
preferences
primes
print, printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
propertyeditor
psi
publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quadl
quadv
questdlg
quit
Quit (COM)
quiver
quiver3
Quivergroup Properties
qz
rand
randn
randperm
rank
rat, rats
rbbox
rcond
readasync
real
reallog
realmax
realmin
realpow
realsqrt
record
rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp, regexpi
regexprep
regexptranslate
registerevent
rehash
release
rem
removets
rename
repmat
resample (timeseries)
resample (tscollection)
reset
reshape
residue
restoredefaultpath
rethrow
return
rgb2hsv
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)
save (serial)
saveas
saveobj
savepath
scatter
scatter3
Scattergroup Properties
schur
script
sec
secd
sech
selectmoveresize
semilogx, semilogy
send
sendmail
serial
serialbreak
set
set (COM)
set (serial)
set (timer)
set (timeseries)
set (tscollection)
setabstime (timeseries)
setabstime (tscollection)
setappdata
setdiff
setenv
setfield
setinterpmethod
setpixelposition
setpref
setstr
settimeseriesnames
setxor
shading
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size (serial)
size (timeseries)
size (tscollection)
slice
smooth3
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
symbfact
symmlq
symmmd
symrcm
symvar
synchronize
syntax
system
tan
 tand
 tanh
 tar
 tempdir
 tempname
 tetramesh
 texlabel
 text
 Text Properties
 textread
 textscan
 textwrap
 tic, toc
 timer
 timerfind
 timerfindall
 timeseries
 title
 todatenum
 toeplitz
 toolboxdir
 trace
 transpose (timeseries)
 trapz
 treelayout
 treeplot
 tril
 trimesh
 triplequad
 triplot
 trisurf
 triu
 true
 try
 tscollection
 tsdata.event
tsearch
tsearchn
tsprops
tstool
type
typecast
uibuttongroup
Ub buttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties
ugetdir
ugetfile
ugetpref
uiimport
uimenu
Uimenu Properties
uint8, uint16, uint32, uint64
uiopen
uipanel
Uipanel Properties
uipushtool
Uipushtool Properties
uipofile
uiresume, uiwait
uisave
uisetcolor
uisetfont
uisetpref
uistack
ui toggletool
Uitoggletool Properties
ui toolbar
Uitoolbar Properties
undocheckout
unicode2native
union
unique
unix
unloadlibrary
unmkpp
unregisterallevents
unregisterevent
untar
unwrap
unzip
upper
urllib
urllib2
usejava
vander
var
var (timeseries)
varargin
varargout
vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (timeseries)
vertcat (tscollection)
view
viewmtx
volumebounds
vorono
voronoin
wait
waitbar
waitfor
waitforbuttonpress
warndlg
warning
waterfall
wavinfo
wavplay
wavread
wavrecord
wavwrite
web
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, []).

MATLAB 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

2-2308
• 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:

• UNIX: Ask your system manager to increase your swap space.
• Windows: 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**
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
Pan view of graph interactively

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 — Moving Your View of the Graph” in the MATLAB Graphics documentation.

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

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. This read-only property cannot be set.

ButtonDownFilter <function_handle>

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 callbacks):

```matlab
function [res] = myfunction(obj,event_obj)
% obj handle to the object that has been clicked on.
% event_obj handle to event object (empty in this release).
% res 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>

Set this callback to listen to when a pan operation will start. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

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

The event object has the following read-only property:

<table>
<thead>
<tr>
<th>Axes</th>
<th>The handle of the axes that is being panned</th>
</tr>
</thead>
</table>

ActionPostCallback <function_handle>

Set this callback to listen to when a pan operation has finished. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

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

flags = isAllowAxesPan(h,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)
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)
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'.

setAxesPanMotion(h,axes,style)
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

Simple pan:

plot(1:10);
pan on
% pan on the plot
Example 2

Constrain pan to x-axis using set:

```matlab
plot(1:10);
h = pan;
set(h,'Motion','horizontal','Enable','on');
% pan on the plot in the horizontal direction.
```

Example 3

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

```matlab
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

Create a buttonDown 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.
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 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.');
%
function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%2f %2f].',newLim));
```
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.

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 1-99 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)
pareto(Y,names)
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.

See Also

hist, bar
### Purpose
Parse and validate named inputs

### Syntax
- `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.

### Note
For more information on the `inputParser` class, see Parsing Inputs with `inputParser` in the MATLAB Programming 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`:

```
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:

```
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:

```
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('The maximum height is %d.
 ', p.\text{Results}\.maxHeight))

% Display all arguments.
disp 'List of all arguments:'
disp(p.\text{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:

```matlab
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)
**parseSoapResponse**

**Purpose**  
Convert response string from SOAP server into MATLAB data types

**Syntax**  
`parseSoapResponse(response)`

**Description**  
`parseSoapResponse(response)` converts `response`, a string returned by a SOAP server, into a cell array of appropriate MATLAB data types.

**Example**
```matlab
message = createSoapMessage(...  
'urn:xmethods-delayed-quotes','getQuote',{'GOOG'},{'symbol'},...  
{'http://www.w3.org/2001/XMLSchema}string'},'rpc')
response = callSoapService('http://64.124.140.30:9090/soap',...  
'urn:xmethods-delayed-quotes#getQuote',message)
price = parseSoapResponse(response)
```

**See Also**  
callSoapService, createClassFromWsdl, createSoapMessage

2-2324
Purpose

Partial pathname description

Description

A partial pathname is a pathname relative to the MATLAB path, `matlabpath`. It is used to locate private and method files, which are usually hidden, or to restrict the search for files when more than one file with the given name exists.

A partial pathname contains the last component, or last several components, of the full pathname separated by `/`. For example, `matfun/trace`, `private/children`, and `demos/clown.mat` are valid partial pathnames. Specifying the `@` in method directory names is optional.

Partial pathnames make it easy to find a toolbox or MATLAB relative files 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

- `help`, `type`, `load`, `exist`, `what`, `which`, `edit`, `dbtype`,
- `dbstop`, `dbclear`, `fopen`

Examples

The following example uses a partial pathname:

```matlab
what graph2d/@figobj
```

M-files in directory

```matlab
matlabroot\toolbox\matlab\graph2d\@figobj
deselectall   enddrag   middrag   subsref
doclick       figobj    set
doreshape     get        subsasgn
```

P-files in directory

```matlab
matlabroot\toolbox\matlab\graph2d\@figobj
deselectall   enddrag   middrag   subsref
```
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

\[
\begin{align*}
A &= \text{pascal}(n) \\
A &= \text{pascal}(n,1) \\
A &= \text{pascal}(n,2)
\end{align*}
\]

Description

\(A = \text{pascal}(n)\) 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.

\(A = \text{pascal}(n,1)\) 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.

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

Examples

\(\text{pascal}(4)\) returns

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

\(A = \text{pascal}(3,2)\) produces

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

See Also

chol

\[2-2327\]
**Purpose**

Create patch graphics object

**Syntax**

\[
\text{patch}(X,Y,C) \\
\text{patch}(X,Y,Z,C) \\
\text{patch}(\text{FV}) \\
\text{patch}(\ldots'\text{PropertyName}',\text{propertyvalue}\ldots) \\
\text{patch('\text{PropertyName}',\text{propertyvalue},\ldots) \\
\text{handle} = \text{patch}(\ldots)
\]

**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” for more information on using patch objects.

\text{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 “Remarks” on page 2-2329). If \(C\) is a 1-by-3 vector, it is assumed to be an RGB triplet, specifying a color directly.

\text{patch}(X,Y,Z,C)\) creates a patch in three-dimensional coordinates.

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

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

\text{patch('\text{PropertyName}',\text{propertyvalue},\ldots)\) 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 the “Examples” on page 2-2332 section for more information.

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 break up 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 ($XData$, $YData$, $ZData$).
- **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.
### Interpretation of the CData Property

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Indexed</th>
<th>True Color</th>
<th>Results Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-by-n</td>
<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>m-by-n</td>
<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</td>
<td>m-by-n</td>
<td>m-by-n-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>

### Interpretation of the FaceVertexCData Property

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Faces</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

This example creates 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 defines two triangular faces, each having three vertices. Using true color, the top face is set to white and the bottom face to gray.

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

Notice that each face shares two vertices with the other face ($V_1$-$V_4$ and $V_3$-$V_5$).
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.

\[
\text{fac} = [1 \ 2 \ 3; 1 \ 3 \ 4];
\]

To specify the face colors, define a 2-by-3 matrix containing two RGB color definitions.

\[
\text{tcolor} = [1 \ 1 \ 1; .7 \ .7 \ .7];
\]

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.

\[
\text{patch}('\text{Faces}', \text{fac}, '\text{Vertices}', \text{vert}, '\text{FaceVertexCData}', \text{tcolor}, \ldots '\text{FaceColor}', '\text{flat}')
\]
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 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.

Object Hierarchy

Setting Default Properties

You can set default patch properties on the axes, figure, and root 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 Functions” on page 1-93 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.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaDataMapping</td>
<td>none</td>
</tr>
</tbody>
</table>

*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 `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 second.
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.

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
functional 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
**Patch Properties**

*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`.

\[
\text{patch}(x,y,z,c,'CreateFcn',@myCreateFcn)
\]

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 `gcbo`.

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.

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:

![Vertex controlling the color of the following edge](image)

- **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

- **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 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**

{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.

• **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
**Patch Properties**

**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 \texttt{Vertices} property, which specifies one color per vertex

For true colors, \texttt{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 \texttt{Faces} property, which specifies one color per face
• An \( n \)-by-3 matrix, where \( n \) is the number of rows in the \texttt{Vertices} property, which specifies one color per vertex

The following diagram illustrates the various forms of the \texttt{FaceVertexCData} property for a patch having eight faces and nine vertices. The \texttt{CDataMapping} property determines how MATLAB interprets the \texttt{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.
Patch Properties

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.
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>
</tbody>
</table>
## Patch Properties

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.
- 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 = $\frac{1}{72}$ inch). Note that MATLAB draws the point marker at $\frac{1}{3}$ 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.
**Patch Properties**

**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.

**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

\[
\text{patch}(X,Y,'k','Tag','PatchBorder')
\]

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

\[
\text{set}(\text{findobj}('Tag','PatchBorder'),'FaceColor','w')
\]

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 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 MATLAB directory search path

GUI
As an alternative to the path function, select File > Set Path to 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 initial search path list is defined by toolbox/local/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.
Note Save any M-files you create and any MathWorks supplied M-files that you edit in a directory that is not in the `matlabroot/toolbox` directory tree. If you keep your files in `matlabroot/toolbox` directories, they can be overwritten when you install a new version of MATLAB. Also note that locations of files in the `matlabroot/toolbox` directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to `matlabroot/toolbox` directories using an external editor or add or remove files from these directories using file system operations, run `rehash toolbox` before you use the files in the current session. If you make changes to existing files in `matlabroot/toolbox` directories using an external editor, run `clear functionname` before you use the files in the current session. For more information, see the `rehash` reference page or the Toolbox Path Caching topic in the MATLAB Desktop Tools and Development Environment documentation.

Examples

Add a new directory to the search path on Windows.

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

Add a new directory to the search path on UNIX.

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

See Also

`addpath`, `cd`, `dir`, `genpath`, `matlabroot`, `partialpath`, `pathdef`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `rmpath`, `savepath`, `startup`, `what`

Search Path topic in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**  
Save current MATLAB 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.
Purpose
Directories in MATLAB search path

GUI
Alternatives
As an alternative to the pathdef function, select File > Set Path to use the Set Path dialog box.

Syntax
pathdef

Description
pathdef returns a string listing of the directories in the MATLAB search path. Use path to view each directory in pathdef.m on a separate line.

When you start a new session, MATLAB creates the search path defined in the pathdef.m file located in the MATLAB startup directory. If that directory does not contain a pathdef.m file, MATLAB uses the search path defined in matlabroot/toolbox/local/pathdef.m. It modifies the search path using any path statements contained in the startup.m file.

Make changes to the path using the Set Path dialog box and addpath and rmpath. While you can edit pathdef.m directly, use caution so you do not accidentally make MATLAB supplied directories unusable. Use savepath to save pathdef.m, and to use that path in future sessions, specify the MATLAB startup directory as its location.

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

MATLAB Desktop Tools and Development Environment documentation topics

- How MATLAB Finds the Search Path, pathdef.m
- Saving Settings to the Path
- Using the Path in Future Sessions
- Recovering from Problems with the Search Path
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 string returned by the `matlabpath` function.

Examples  Extract each individual path from the string returned by `matlabpath`. Use `pathsep` to define the path separator:

```matlab
s = matlabpath;
p = 1;

while true
    t = strtok(s(p:end), pathsep);
    disp(sprintf('%s', t))
    p = p + length(t) + 1;
    if isempty(strfind(s(p:end),';')) break, end;
end
```

Here is the output:

```
D:\Applications\matlabR14beta2\toolbox\matlab\general
D:\Applications\matlabR14beta2\toolbox\matlab\ops
D:\Applications\matlabR14beta2\toolbox\matlab\lang
D:\Applications\matlabR14beta2\toolbox\matlab\elmat
D:\Applications\matlabR14beta2\toolbox\matlab\elfun
```

See Also  `filesep`, `fullfile`, `fileparts`
**Purpose**
Open Set Path dialog box to view and change MATLAB 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 MATLAB search path.

---

**Set Path**

- **Add Folder...**
- **Add with Subfolders...**
- **Move to Top**
- **Move Up**
- **Move Down**
- **Move to Bottom**
- **Remove**

**Directories on the current search path**

**Make changes to the search path.**

**Save changes for use in future MATLAB sessions.**

**Use the changes for the current session, but do not save the changes for use in future MATLAB sessions.**

**Replace current path with all directories installed with MATLAB and related products.**

**Save**
**Close**
**Revert**
**Default**
**Help**
See Also

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

Search Path topics, including Setting the Search Path, in the MATLAB Desktop Tools and Development Environment documentation
Purpose
Halt execution temporarily

Syntax
pause
pause(n)
pause on
pause off

Description
pause, by itself, causes M-files to stop and wait for you to press any key before continuing.
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.

Typing `pause(inf)` puts you into an infinite loop. To return to the MATLAB prompt, type `Ctrl+C`.
pause on allows subsequent pause commands to pause execution.
pause off ensures that any subsequent pause or `pause(n)` statements do not pause execution. This allows normally interactive scripts to run unattended.

Remarks
While MATLAB is paused, the following continue to execute:

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

When MATLAB is paused and a uicontrol has focus, pressing a keyboard key does not cause MATLAB to resume. You can resume your MATLAB session by clicking anywhere outside the uicontrol, and then pressing any key. Uicontrols include check boxes, editable text fields, list boxes, pop-up menus, push buttons, radio buttons, sliders, static text labels, and toggle buttons.
See Also
drawnow
Purpose
Set or query plot box aspect ratio

Syntax

```matlab
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, MATLAB 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 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,

\[ \text{pbaspect(pbaspect)} \]

can cause a change in the way the graphs look. See the Remarks section of the 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 \(-2 \leq x \leq 2, -2 \leq y \leq 2\),

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

    pbaspect
    ans =
       1   1   1

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

    daspect
    ans =
       4   4   1

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.

    daspect([1 1 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.

\[
\text{pbaspect([1 1 1])}
\]
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
\[
x = \text{pcg}(A,b)
\]
\[
\text{pcg}(A,b,tol)
\]
\[
\text{pcg}(A,b,tol,maxit)
\]
\[
\text{pcg}(A,b,tol,maxit,M)
\]
\[
\text{pcg}(A,b,tol,maxit,M1,M2)
\]
\[
\text{pcg}(A,b,tol,maxit,M1,M2,x0)
\]
\[
[x,\text{flag}] = \text{pcg}(A,b,...)
\]
\[
[x,\text{flag},\text{relres}] = \text{pcg}(A,b,...)
\]
\[
[x,\text{flag},\text{relres},\text{iter}] = \text{pcg}(A,b,...)
\]
\[
[x,\text{flag},\text{relres},\text{iter},\text{resvec}] = \text{pcg}(A,b,...)
\]

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.

“Parameterizing Functions Called by Function 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 \(\frac{\text{norm}(b-A*x)}{\text{norm}(b)}\) and the iteration number at which the method stopped or failed.

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

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

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

\[
\text{pcg}(A,b,\text{tol},\text{maxit},M1,M2,x0)
\]
specifies the initial guess. If \( x0 \) is \([\,]\), then 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>pcg converged to the desired tolerance ( \text{tol} ) within ( \text{maxit} ) iterations.</td>
</tr>
<tr>
<td>1</td>
<td>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>pcg stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during pcg became too small or too large to continue computing.</td>
</tr>
</tbody>
</table>

Whenever \( \text{flag} \) is not \( 0 \), the solution \( \mathbf{x} \) 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 \( \frac{\text{norm}(b-A\mathbf{x})}{\text{norm}(b)} \). 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 \( \mathbf{x} \) 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 \( \text{norm}(b-A\mathbf{x}0) \).

**Examples**

**Example 1**

\[
\begin{align*}
n1 &= 21; \\
A &= \text{gallery}(\text{moler},n1); \\
b1 &= A*\text{ones}(n1,1); \\
tol &= 1\text{e}-6;
\end{align*}
\]
maxit = 15;
M = diag([10:-1:1 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**

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

**Description**

\( yi = \text{pchip}(x,y,xi) \) returns vector \( yi \) 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 \( yi \) is \( \text{length}(xi) \)-by-\( \text{size}(y,2) \).

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

**pcode**

**Purpose**
Create preparsed pseudocode file (P-file)

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

**Description**
- `pcode fun` parses the M-file `fun.m` into the P-file `fun.p` and puts it into the current directory. The original M-file can be anywhere on the search path.
- `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 can’t be created.
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

\[
\text{pcolor}(C) \\
\text{pcolor}(X,Y,C) \\
\text{pcolor}(\text{axes\_handles},...) \\
h = \text{pcolor}(...) \\
\]

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.

\( \text{pcolor}(C) \) 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.

\( \text{pcolor}(X,Y,C) \) 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.

\texttt{pcolor(axes\_handles,...)} plots into the axes with handle \texttt{axes\_handle} instead of the current axes (\texttt{gca}).

\texttt{h = pcolor(...)} returns a handle to a surface graphics object.

**Remarks**

A pseudocolor plot is a flat surface plot viewed from above. \texttt{pcolor(X,Y,C)} is the same as viewing \texttt{surf(X,Y,zeros(size(X)),C)} using \texttt{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 \(i\)th row and \(j\)th 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.

\begin{verbatim}
    pcolor(hadamard(20))
    colormap(gray(2))
    axis ij
    axis square
\end{verbatim}
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**  
\[
sol = \text{pdepe}(m, \text{pdefun}, \text{icfun}, \text{bcfun}, \text{xmesh}, \text{tspan})
\]
\[
sol = \text{pdepe}(m, \text{pdefun}, \text{icfun}, \text{bcfun}, \text{xmesh}, \text{tspan}, \text{options})
\]

**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 ( \text{tspan} ). The elements of ( \text{xmesh} ) must satisfy ( x0 &lt; x1 &lt; ... &lt; xn ). The length of ( \text{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 ( \text{xmesh} ). The elements of ( \text{tspan} ) must satisfy ( t0 &lt; t1 &lt; ... &lt; tf ). The length of ( \text{tspan} ) must be ( &gt;= 3 ).</td>
</tr>
<tr>
<td>options</td>
<td>Some options of the underlying ODE solver are available in ( \text{pdepe} ): RelTol, AbsTol, NormControl, InitialStep, and MaxStep. In most cases, default values for these options provide satisfactory solutions. See ( \text{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 \( \mathbf{x} \) and time \( t \). \( \text{pdefun} \), \( \text{icfun} \), 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.

“Parameterizing Functions Called by Function Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the functions pdefun, 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

\[ u(x, t_0) = u_0(x) \]
For all $t$ and either $x = a$ or $x = b$, the solution components satisfy a boundary condition of the form

$$p(x, t, u) + q(x, t) f(x, t, u, \frac{\partial u}{\partial x}) = 0$$

(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 $sol = \text{pdepe}(m, \text{pdefun}, \text{icfun}, \text{bcfun}, \text{xmesh}, \text{tspan})$:

- $m$ 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

  $$u = \text{icfun}(x)$$

  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

  $$[pl, ql, pr, qr] = \text{bcfun}(xl, ul, xr, ur, t)$$
pdepe

u_l is the approximate solution at the left boundary x_l = a and u_r is the approximate solution at the right boundary x_r = b. p_l and q_l are column vectors corresponding to P and Q evaluated at x_l, similarly p_r and q_r correspond to x_r. When m > 0 and a = 0, boundedness of the solution near \( x = 0 \) requires that the flux \( f \) vanish at \( a = 0 \). pdepe imposes this boundary condition automatically and it ignores values returned in p_l and q_l.

pdepe returns the solution as a multidimensional array sol.

\( u_i = u_i = \text{sol}(:, :, i) \) is an approximation to the i-th component of the solution vector \( u \). The element \( u_i(j, k) = \text{sol}(j, k, i) \) approximates \( u_i \) at \( (t, x) = (tspan(j), xmesh(k)) \).

\( u_i = \text{sol}(j, :, i) \) approximates component \( i \) of the solution at time \( tspan(j) \) and mesh points \( xmesh(\cdot) \). Use pdeval to compute the approximation and its partial derivative \( \frac{\partial u_i}{\partial x} \) 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 \( \text{options} \), 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 \( \text{options} \) 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
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 \texttt{ode15s}. \texttt{pdepe} exploits the capabilities of \texttt{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, \texttt{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, \texttt{pdepe} can find consistent initial conditions close to the given ones. If \texttt{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.

\[
\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left( \frac{\partial u}{\partial x} \right)
\]

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);
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
```
\[
pr = \pi \cdot \exp(-t);
qr = 1;
\]

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 $t$. 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.

```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);
```
\[ F = \exp(5.73y) - \exp(-11.47y); \]
\[ s = [-F; F]; \]

\begin{verbatim}
function u0 = pdex4ic(x);
u0 = [1; 0];
\end{verbatim}

\begin{verbatim}
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];
\end{verbatim}

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

**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><code>m</code></td>
<td>Symmetry of the problem: slab = 0, cylindrical = 1, spherical = 2. This is the first input argument used in the call to <code>pdepe</code>.</td>
</tr>
<tr>
<td><code>xmesh</code></td>
<td>A vector <code>[x0, x1, ..., xn]</code> specifying the points at which the elements of <code>ui</code> were computed. This is the same vector with which <code>pdepe</code> was called.</td>
</tr>
<tr>
<td><code>ui</code></td>
<td>A vector <code>sol(j,:,i)</code> that approximates component <code>i</code> of the solution at time <code>t_f</code> and mesh points <code>xmesh</code>, where <code>sol</code> is the solution returned by <code>pdepe</code>.</td>
</tr>
<tr>
<td><code>xout</code></td>
<td>A vector of points from the interval <code>[x0,xn]</code> at which the interpolated solution is requested.</td>
</tr>
</tbody>
</table>

**Description**

`[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution `u_i` and its partial derivative `\frac{\partial u_i}{\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_i}{\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

![Peaks](image)

**Syntax**

\[
\begin{align*}
Z &= \text{peaks;} \\
Z &= \text{peaks}(n); \\
Z &= \text{peaks}(V); \\
Z &= \text{peaks}(X,Y);
\end{align*}
\]

peaks;
peaks(N);
peaks(V);
peaks(X,Y);

\[
\begin{align*}
X,Y,Z] &= \text{peaks;} \\
[X,Y,Z] &= \text{peaks}(n); \\
[X,Y,Z] &= \text{peaks}(V);
\end{align*}
\]

**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.

\[
\begin{align*}
Z &= \text{peaks;} \text{ returns a 49-by-49 matrix.} \\
Z &= \text{peaks}(n); \text{ returns an n-by-n matrix.} \\
Z &= \text{peaks}(V); \text{ returns an n-by-n matrix, where } n = \text{length}(V). \\
Z &= \text{peaks}(X,Y); \text{ evaluates peaks at the given X and Y (which must be the same size) and returns a matrix the same size.}
\end{align*}
\]
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(...)  
```

**Description**

`perl('perlfile')` calls the Perl script `perlfile`, using the appropriate operating system Perl executable. Perl is included with MATLAB on Windows systems, and thus MATLAB users can run M-files containing the `perl` function. On Unix systems, MATLAB just calls the Perl interpreter that’s available with the OS.

`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`.

**Examples**

Given the Perl script, `hello.pl`

```
$input = $ARGV[0];
print "Hello $input.");
```

run the following statement in MATLAB

```
perl('hello.pl','World')
```

MATLAB returns

```
ans =
Hello World.
```

It is sometimes beneficial to use Perl scripts instead of MATLAB code. The `perl` function allows you to run those scripts from within 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

**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

\text{nchoosek}, \text{permute}, \text{randperm}
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
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 MATLAB 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.

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:

```matlab
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/

isempty.m isfinite.m isscalar.m
isequal.m isinf.m isvector.m
isequalwithequalnans.m isnan.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**

```plaintext
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,

```plaintext
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 $X / \sum(X)$ to determine the area of each slice of the pie. If $\sum(X) \leq 1$, the values in X directly specify the area of the pie slices. MATLAB draws only a partial pie if $\sum(X) < 1$.

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:

\[
\begin{align*}
A*B*A &= A \\
B*A*B &= B \\
A*B &\text{ is Hermitian} \\
B*A &\text{ is Hermitian}
\end{align*}
\]

The computation is based on $\text{svd}(A)$ and any singular values less than $\text{tol}$ are treated as zero.

**Description**

$B = \text{pinv}(A)$ returns the Moore-Penrose pseudoinverse of $A$.

$B = \text{pinv}(A,tol)$ 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 $\text{pinv}(A)$ is an expensive way to compute $\text{inv}(A)$. If $A$ is not square, or is square and singular, then $\text{inv}(A)$ does not exist. In these cases, $\text{pinv}(A)$ has some of, but not all, the properties of $\text{inv}(A)$.

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

\[
x = \text{pinv}(A)*b
\]

and

\[
y = A\backslash b
\]
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

\[
A = \text{magic}(8); \ A = A(:,1:6)
\]

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

\[
\begin{bmatrix}
1.1538 \\
1.4615 \\
1.3846 \\
1.3846 \\
1.4615 \\
1.1538
\end{bmatrix}
\]

and

\[
y = A \backslash b
\]

which produces this result.

\[
\text{Warning: Rank deficient, rank = 3 \ tol = 1.8829e-013.}
\]

\[
y = \\
\begin{bmatrix}
4.0000 \\
5.0000 \\
0 \\
0 \\
0 \\
-1.0000
\end{bmatrix}
\]

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 \(y = Gx\) has \(y(2) = 0\).

Examples  
\[
x = [3 4];
\]
\[
[G,y] = \text{planerot}(x')
\]

\[
G = \begin{bmatrix}
0.6000 & 0.8000 \\
-0.8000 & 0.6000
\end{bmatrix}
\]

\[
y = \begin{bmatrix}
5 \\
0
\end{bmatrix}
\]

See Also  qrdelete, qrinsert
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Run M-file demo (deprecated; use echodemo 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

GUI
Use the Plot Selector 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
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineSpec,...)
plot(...,'PropertyName',PropertyValue,...)
plot(axes_handle,...)
h = 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 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.

plot(...,'PropertyName',PropertyValue,...) sets properties to the specified property values for all lineseries graphics objects created by `plot`. (See the “Examples” on page 2-2420 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.

**Remarks**

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 axes `LineStyleOrder` property.

The default `LineStyleOrder` property has a single entry (a solid line with no marker).

**Cycling Through Line Colors and Styles**

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','-|-.|--|:');
```
plot

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:
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/4)',...}
```
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 1-85 for related functions.
Purpose
Plot time series

Syntax
plot(ts)
plot(tsc.tsname)
plot(function)

Description
plot(ts) plots the time-series data 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.

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,...)
```

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 X1, Y1, Z1 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 Xn, Yn, Zn triples with Xn, Yn, Zn, *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
```
plot3

See Also

axis, bar3, grid, line, LineSpec, loglog, plot, semilogx, semilogy, subplot
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>on</td>
<td>Starts plot edit mode</td>
</tr>
<tr>
<td>off</td>
<td>Ends plot edit mode</td>
</tr>
<tr>
<td>showtoolsmenu</td>
<td>Displays the <strong>Tools</strong> menu in the menu bar</td>
</tr>
<tr>
<td>hidetoolsmenu</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(...,'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(Y) is the same as plotmatrix(Y,Y) except that the diagonal is replaced by hist(Y(:,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
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

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,

    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.

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:

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:

xlabel('Time ('musec')')
title('Multiple Decay Rates')

Use the line handles to set the LineStyle properties of the left- and
right-side plots:

set(H1,'LineStyle','--')
set(H2,'LineStyle',':')
See Also

plot, loglog, semilogx, semilogy, axes properties XAxisLocation, YAxisLocation

See “Using Multiple X- and Y-Axes” for more information.
**Purpose**
Transform polar or cylindrical coordinates to Cartesian

**Syntax**

\[ [X,Y] = \text{pol2cart}(\text{THETA},\text{RHO}) \]
\[ [X,Y,Z] = \text{pol2cart}(\text{THETA},\text{RHO},Z) \]

**Description**

\[ [X,Y] = \text{pol2cart}(\text{THETA},\text{RHO}) \] transforms the polar coordinate data stored in corresponding elements of \( \text{THETA} \) and \( \text{RHO} \) to two-dimensional Cartesian, or \( xy \), coordinates. The arrays \( \text{THETA} \) and \( \text{RHO} \) must be the same size (or either can be scalar). The values in \( \text{THETA} \) must be in radians.

\[ xyz, [X,Y,Z] = \text{pol2cart}(\text{THETA},\text{RHO},Z) \] transforms the cylindrical coordinate data stored in corresponding elements of \( \text{THETA}, \text{RHO}, \) and \( Z \) to three-dimensional Cartesian, or \( xyz \), coordinates. The arrays \( \text{THETA}, \text{RHO}, \) and \( Z \) must be the same size (or any can be scalar). The values in \( \text{THETA} \) must be in radians.

**Algorithm**
The mapping from polar and cylindrical coordinates to Cartesian coordinates is:

\[ \theta = \text{atan2}(y,x) \]
\[ \rho = \sqrt{x.^2 + y.^2} \]

\[ \theta = \text{atan2}(y,x) \]
\[ \rho = \sqrt{x.^2 + y.^2} \]
\[ z = z \]
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**

```matlab
polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axces_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(axces_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
Polyynomial 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, \(\det(sI - A)\). 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

\[
A =
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 0
\end{bmatrix}
\]

is returned in a row vector by \text{poly}:

\[
p = \text{poly}(A)
\]

\[
p =
\begin{bmatrix}
2 & -24 & 44
\end{bmatrix}
\]
The roots of this polynomial (eigenvalues of matrix A) are returned in a column vector by `roots`:

\[ r = \text{roots}(p) \]

\[ r = \begin{bmatrix} 12.1229 \\ -5.7345 \\ -0.3884 \end{bmatrix} \]

**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

\[
det(\lambda I - A) = c_1\lambda^n + \ldots + c_n\lambda + c_{n+1}
\]

The algorithm is

```matlab
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
for j = 1:n
    c(2:j+1) = c(2:j+1)-z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product.

\[
(\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n)
\]
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**

\[ A = \text{polyarea}(X,Y) \]

\[ A = \text{polyarea}(X,Y,\text{dim}) \]

**Description**

\( A = \text{polyarea}(X,Y) \) 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 \( Y \).

\( A = \text{polyarea}(X,Y,\text{dim}) \) operates along the dimension specified by scalar \( \text{dim} \).

**Examples**

\begin{verbatim}
L = linspace(0,2.*pi,6); xv = cos(L)'; yv = sin(L)';
xv = [xv; xv(1)]; yv = [yv; yv(1)];
A = polyarea(xv,yv);
plot(xv,yv); title(['Area = ' num2str(A)]); axis image
\end{verbatim}
polyarea

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

a = [3 6 9];
b = [1 2 0];
k = polyder(a,b)
k =

\[12 \quad 36 \quad 42 \quad 18\]

This result represents the polynomial

\[12x^3 + 36x^2 + 42x + 18\]

See Also
conv, deconv
**Purpose**

Polynomial eigenvalue problem

**Syntax**

\[
\begin{align*}
[X, e] &= \text{polyeig}(A_0, A_1, \ldots, A_p) \\
e &= \text{polyeig}(A_0, A_1, \ldots, A_p) \\
[X, e, s] &= \text{polyeig}(A_0, A_1, \ldots, A_P)
\end{align*}
\]

**Description**

\[ [X, e] = \text{polyeig}(A_0, A_1, \ldots, A_p) \] 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 \( A_0, A_1, \ldots, A_p \) are input matrices of order \( n \). The output consists of a matrix \( X \) of size \( n \times n \times p \) whose columns are the eigenvectors, and a vector \( e \) of length \( n \times p \) containing the eigenvalues.

If \( \lambda \) is the \( j \)th eigenvalue in \( e \), and \( x \) is the \( j \)th column of eigenvectors in \( X \), then \( (A_0 + \lambda A_1 + \ldots + \lambda^p A_p)x \) is approximately 0.

\[ e = \text{polyeig}(A_0, A_1, \ldots, A_p) \] is a vector of length \( n \times p \) whose elements are the eigenvalues of the polynomial eigenvalue problem.

\[ [X, e, s] = \text{polyeig}(A_0, A_1, \ldots, A_P) \] also returns a vector \( s \) of length \( p \times n \) containing condition numbers for the eigenvalues. At least one of \( A_0 \) and \( A_P \) 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 \( n \), \text{polyeig} handles several special cases:

- \( p = 0 \), or \text{polyeig}(A) is the standard eigenvalue problem: \text{eig}(A).
- \( p = 1 \), or \text{polyeig}(A,B) is the generalized eigenvalue problem: \text{eig}(A,-B).
- \( n = 1 \), or \text{polyeig}(a0,a1,...,ap) for scalars \( a_0, a_1 \ldots, a_p \) is the standard polynomial problem: \text{roots([a_p \ldots a_1 a_0])}.
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 \texttt{polyeig}. In general, however, this cannot be achieved. (See Tisseur [3] for more detail.)

**Algorithm**

The \texttt{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 \texttt{eig} and \texttt{qz} for more on this.

**See Also**

\texttt{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

\[ p(x) = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_n + ] \]

\( [p,S] = \text{polyfit}(x,y,n) \) returns the polynomial coefficients \( p \) and a structure \( S \) for use with \( \text{polyval} \) to obtain error estimates or predictions. Structure \( S \) contains fields \( R, \, \text{df}, \) 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

\[ (Rinv*Rinv')*\text{normr}^2/\text{df} \]

where \( Rinv \) 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

\[
0.0084 x^6 - 0.0983 x^5 + 0.4217 x^4 - 0.7435 x^3 + 0.1471 x^2 + 1.1064 x + 0.0004
\]

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

\[
table = [x \ y \ f \ y-f]
table =
\begin{bmatrix}
0 & 0 & 0.0004 & -0.0004 \\
0.1000 & 0.1125 & 0.1119 & 0.0006 \\
0.2000 & 0.2227 & 0.2223 & 0.0004 \\
0.3000 & 0.3286 & 0.3287 & -0.0001 \\
0.4000 & 0.4284 & 0.4288 & -0.0004 \\
\ldots \\
2.1000 & 0.9970 & 0.9969 & 0.0001 \\
2.2000 & 0.9981 & 0.9982 & -0.0001 \\
2.3000 & 0.9989 & 0.9991 & -0.0003 \\
2.4000 & 0.9993 & 0.9995 & -0.0002
\end{bmatrix}
\]
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 \( x \).
\[ u_{i,j} = x_i^{n-j} \]

It then uses the backslash operator, \( \backslash \), to solve the least squares problem

\[ Vp = y \]

You can modify the M-file to use other functions of \( x \) as the basis functions.

**See Also**

poly, polyval, roots
**Purpose**
Integrate polynomial analytically

**Syntax**
polyint(p,k)
polyint(p)

**Description**
polyint(p,k) returns a polynomial representing the integral of polynomial p, using a scalar constant of integration k.
polyint(p) assumes a constant of integration k=0.

**See Also**
polyder, polyval, polyvalm, polyfit
**Purpose**

Polynomial evaluation

**Syntax**

\[
y = \text{polyval}(p, x) \\
y = \text{polyval}(p, x, [], mu) \\
[y, \delta] = \text{polyval}(p, x, S) \\
[y, \delta] = \text{polyval}(p, x, S, mu)
\]

**Description**

\[ y = \text{polyval}(p, x) \] 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.

\[ y = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_n + ] \]

\( x \) can be a matrix or a vector. In either case, \( \text{polyval} \) evaluates \( p \) at each element of \( x \).

\( y = \text{polyval}(p, x, [], mu) \) 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 \( \mu = [\mu_1, \mu_2] \) are optional output computed by \( \text{polyfit} \).

\[ [y, \delta] = \text{polyval}(p, x, S) \] and \( [y, \delta] = \text{polyval}(p, x, S, mu) \) use the optional output structure \( S \) generated by \( \text{polyfit} \) to generate error estimates, \( y \pm \delta \). If the errors in the data input to \( \text{polyfit} \) are independent normal with constant variance, \( y \pm \delta \) contains at least 50% of the predictions.

**Remarks**

The \( \text{polyvalm}(p, x) \) function, with \( x \) a matrix, evaluates the polynomial in a matrix sense. See \( \text{polyvalm} \) 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.

\[
X = \text{pascal}(4)
\]

\[
X = \\
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 20
\]

Its characteristic polynomial can be generated with the `poly` function.

\[
p = \text{poly}(X)
\]

\[
p = \\
1 & -29 & 72 & -29 & 1
\]

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.

\[
\text{polyval}(p,X)
\]

\[
\text{ans} = \\
16 & 16 & 16 & 16 \\
16 & 15 & -140 & -563 \\
16 & -140 & -2549 & -12089
\]
But evaluating it in a matrix sense is interesting.

\[
\begin{pmatrix}
16 & -563 & -12089 & -43779 \\
\end{pmatrix}
\]

\[\text{polyvalm}(p,X)\]
\[
\begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

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 = pow2(Y)
X = pow2(F,E)

Description
X = pow2(Y) returns an array X whose elements are 2 raised to the power Y.

X = 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.

\[ C = \text{power}(A,B) \] is called for the syntax ‘\( A.^B \)’ when \( A \) or \( B \) is an object.

Note that if the \( \text{abs}(Y) \) is less than one, the \text{power} function returns the complex roots. To obtain the remaining real roots, use the \text{nthroot} function.

**See Also**
\text{nthroot}, \text{realpow}
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Evaluate piecewise polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>v = ppval(pp,xx)</code></td>
</tr>
</tbody>
</table>
| **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 `interp1`, `pchip`, `spline`, or the spline utility `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 `pchip`, `spline`, or `mkpp` using the orientation of non-scalar function values specified for those functions, then:

If $f$ is $[D_1,\ldots,D_r]$-valued, and $xx$ is a vector of length $N$, then $V$ has size $[D_1,\ldots,D_r, N]$, with $V(:,\ldots,:,J)$ the value of $f$ at $xx(J)$. 

If $f$ is $[D_1,\ldots,D_r]$-valued, and $xx$ has size $[N_1,\ldots,N_s]$, then $V$ has size $[D_1,\ldots,D_r, N_1,\ldots,N_s]$, with $V(:,\ldots,:, J_1,\ldots,J_s)$ the value of $f$ at $xx(J_1,\ldots,J_s)$. 

If $pp$ was constructed by `interp1` using the orientation of non-scalar function values specified for that function, then:

If $f$ is $[D_1,\ldots,D_r]$-valued, and $xx$ is a vector of length $N$, then $V$ has size $[N,D_1,\ldots,D_r]$, with $V(J,:,\ldots,:)$ the value of $f$ at $xx(J)$. 

If $f$ is $[D_1,\ldots,D_r]$-valued, and $xx$ has size $[N_1,\ldots,N_s]$, then $V$ has size $[N_1,\ldots,N_s,D_1,\ldots,D_r]$, with $V(J_1,\ldots,J_s,:,\ldots,:)$ the value of $f$ at $xx(J_1,\ldots,J_s)$. |
<table>
<thead>
<tr>
<th><strong>Examples</strong></th>
<th>Compare the results of integrating the function $\cos$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a = 0; \ b = 10; \ int1 = \text{quad(@cos,a,b)}$</td>
</tr>
<tr>
<td></td>
<td>$\text{int1} =$</td>
</tr>
<tr>
<td></td>
<td>$-0.5440$</td>
</tr>
</tbody>
</table>
with the results of integrating the piecewise polynomial pp that approximates the cosine function by interpolating the computed values x and y.

\[
\begin{align*}
x &= a:b; \\
y &= \cos(x); \\
pp &= \text{spline}(x,y); \\
\text{int2} &= \text{quad}(@(x)\text{ppval}(pp,x),a,b)
\end{align*}
\]

\[
\text{int2} = -0.5485
\]

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
d = prefdir
```

**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

The directory might be in a hidden folder, for example, `myname/.matlab/R2007a`. How to access hidden folders depends on your platform:

- On Windows, 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 Macintosh platforms, in the Finder, select **Go -> Go to Folder**. In the resulting dialog box, type the path returned by `prefdir` and press **Enter**.

`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

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 R2007a, R2006b, R2006a, and R14SP3, MATLAB uses the R2007a, R2006b, R2006a, and R14SP3 preferences directories, respectively. When you install R2007a, MATLAB migrates the files in the R2006b preferences directory to the R2007a preferences directory. While running R2007a, R2006b, R2006a, or R14SP3, any changes made to files in those preferences directories (R2007a, R2006b, R2006a, or R14SP3) are used only in their respective versions. As an example, commands you run in R2007a will not appear in the Command History when you run R2006b, R2006a, or R14SP3, and the converse is also true.

- 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, R2006a, R2006b, or R2007a 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.

- Upon startup, MATLAB 7.4 (R2007a) looks for and if found, uses the R2007a preferences directory. If not found, MATLAB creates an R2007a preferences directory. This happens when the R2007a
preferences directory is deleted. MATLAB then looks for the R2006b preferences directory, and if found, migrates the R2006b preferences to the R2007a preferences. If it does not find the R2006b preferences directory, it uses the default preferences for R2007a. The process also applies when MATLAB 7.3, 7.2, and 7.1 versions start.

- If you want to use default preferences for R2007a, and do not want MATLAB to migrate preferences from R2006b, the R2007a preferences directory must exist but be empty when you start MATLAB. If you want to maintain some of your R2007a preferences, but restore the defaults for others, in the R2007a 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.

**Examples**

Run

```
prefdir
```

MATLAB returns

```
ans =
C:\WINNT\Profiles\tbear.MATHWORKS
\Application Data\MathWorks\MATLAB\R2007a
```

Running `dir` for the directory shows the files

```
. history.m
.. matlab.prf
cwdhistory.m MATLABDesktop.xml
shortcuts.xml MATLAB EditorDesktop.xml
```

and possibly other files for MATLAB and other MathWorks products.

In MATLAB, run `cd(prefdir)` to change to that directory.
On Windows platforms, go directly to the preferences directory in Explorer by running `winopen(prefdir)`.

**See Also**

`preferences`, `winopen`

Fonts, Colors, and Other Preferences in the MATLAB Desktop Tools and Development Environment documentation
Purpose
Open Preferences dialog box for MATLAB and related products

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

Fonts, Colors, and Other Preferences in the MATLAB Desktop Tools
and Development Environment documentation
Purpose
Generate list of prime numbers

Syntax
\( p = \text{primes}(n) \)

Description
\( p = \text{primes}(n) \) 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
\( p = \text{primes}(37) \)

\[
p = 2 \ 3 \ 5 \ 7 \ 11 \ 13 \ 17 \ 19 \ 23 \ 29 \ 31 \ 37
\]

See Also
factor
Purpose

Print figure or save to file and configure printer defaults

GUI Alternative

Use **File —> Print** on the figure window menu to access the Print dialog and **File —> Print Preview** to access the Print Preview GUI. For details, see How to Print or Export in the MATLAB Graphics documentation.

Syntax

```
print
print filename
print -ddriver
print -dformat
print -dformat filename
print -smodelname
print -options
print(...)
[pcmd,dev] = printopt
```

Description

print and printopt produce hardcopy 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 -ddriver prints the figure using the specified printer **driver**, (such as color PostScript). If you omit -ddriver, print uses the default value stored in printopt.m. The Printer Driver table lists all supported device types.

print -dformat copies the figure to the system clipboard (Windows only). A valid **format** for this operation is 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 Graphics Format table 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.) The Options section 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 filenames 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 hardcopy 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>System Printing Command</th>
<th>Driver or Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>

**Drivers**

The table below shows the more widely used printer drivers supported by MATLAB. If you do not specify a driver, MATLAB uses the default setting shown in the previous table. For a list of all supported printer drivers, type

    print -d

at the MATLAB prompt.
Some of the drivers are available from a product called Ghostscript, which is shipped with MATLAB. The last column indicates when Ghostscript is used.

Some drivers are not available on all platforms. This is noted in the first column of the table.

<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>Canon BubbleJet BJ200 color</td>
<td>-dbj200</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Canon Color BubbleJet BJC-70/BJC-600/BJC-4000</strong></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><strong>Epson</strong> and compatible 9- or 24-pin dot matrix print drivers</td>
<td>-depson</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Epson</strong> and compatible 9-pin with interleaved lines (triple resolution)</td>
<td>-deps9high</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Epson LQ-2550 and compatible; color (not supported on HP-700)</strong></td>
<td>-depsonc</td>
<td>Yes</td>
</tr>
<tr>
<td>Fujitsu 3400/2400/1200</td>
<td>-depsonc</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP DesignJet 650C color (not supported on Windows)</strong></td>
<td>-ddnj650c</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP DeskJet 500</strong></td>
<td>-ddjet500</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>HP DeskJet 500C</strong></td>
<td>-dcdjmono</td>
<td>Yes</td>
</tr>
<tr>
<td>(creates black and white output)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HP DeskJet 500C</strong></td>
<td>-dcdjcolor</td>
<td>Yes</td>
</tr>
<tr>
<td>(with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HP DeskJet 500C/540C</strong> color (not supported on Windows)</td>
<td>-dcdj500</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP Deskjet 550C</strong> color (not supported on Windows)</td>
<td>-dcdj550</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP DeskJet and DeskJet Plus</strong></td>
<td>-ddeskjet</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP LaserJet</strong></td>
<td>-dlaserjet</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP LaserJet+</strong></td>
<td>-dljetplus</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP LaserJet IIP</strong></td>
<td>-dljet2p</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP LaserJet III</strong></td>
<td>-dljet3</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP LaserJet 4.5L and 5P</strong></td>
<td>-dljet4</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP LaserJet 5 and 6</strong></td>
<td>-dpxlmono</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP PaintJet color</strong></td>
<td>-dpaintjet</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP PaintJet XL color</strong></td>
<td>-dpjxl</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>HP PaintJet XL color</strong></td>
<td>-dpjetx1</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>HP PaintJet XL300</strong></td>
<td>-dpjx1300</td>
<td>Yes</td>
</tr>
<tr>
<td>color (not supported on Windows)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HPGL</strong></td>
<td>-dhp3l</td>
<td>No</td>
</tr>
<tr>
<td>for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IBM 9-pin Proprinter</strong></td>
<td>-dibmpr</td>
<td>Yes</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>-dp3c</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>-dp3c2</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>

**Note** 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 is the default for UNIX. You can change this default by editing the printopt.m file. Likewise, if you want color PostScript to be the default instead of black-and-white PostScript, edit the line in the printopt.m file that reads `dev = '-dps2';` to be `dev = '-dp3c2';`.
To save your figure as a graphics-format file, specify a format switch and filename. 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, and PNG files, but is not supported for Ghostscript formats.

The table below shows the supported output formats for exporting from MATLAB 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 the 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><strong>BMP</strong> monochrome BMP</td>
<td>Bitmap</td>
<td>-dbmpmono</td>
<td>Ghostscript</td>
</tr>
<tr>
<td><strong>BMP</strong> 24-bit BMP</td>
<td>Bitmap</td>
<td>-dbmp16m</td>
<td>Ghostscript</td>
</tr>
<tr>
<td><strong>BMP</strong> 8-bit (256-color) BMP (this format uses a fixed colormap)</td>
<td>Bitmap</td>
<td>-dbmp256</td>
<td>Ghostscript</td>
</tr>
<tr>
<td><strong>BMP</strong> 24-bit</td>
<td>Bitmap</td>
<td>-dbmp</td>
<td>MATLAB</td>
</tr>
<tr>
<td><strong>EMF</strong></td>
<td>Vector</td>
<td>-dmeta</td>
<td>MATLAB</td>
</tr>
<tr>
<td><strong>EPS</strong> black and white</td>
<td>Vector</td>
<td>-deps</td>
<td>MATLAB</td>
</tr>
<tr>
<td><strong>EPS</strong> color</td>
<td>Vector</td>
<td>-depsc</td>
<td>MATLAB</td>
</tr>
<tr>
<td><strong>EPS</strong> Level 2 black and white</td>
<td>Vector</td>
<td>-deps2</td>
<td>MATLAB</td>
</tr>
<tr>
<td><strong>EPS</strong> Level 2 color</td>
<td>Vector</td>
<td>-depsc2</td>
<td>MATLAB</td>
</tr>
<tr>
<td><strong>HDF</strong> 24-bit</td>
<td>Bitmap</td>
<td>-dhdf</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>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 PCX file format, three 8-bit planes</td>
<td>Bitmap</td>
<td>-dpcx24b</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX 8-bit newer color PCX file format (256-color)</td>
<td>Bitmap</td>
<td>-dpcx256</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX Older color PCX 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>
<tr>
<td>PNG 24-bit</td>
<td>Bitmap</td>
<td>-dpng</td>
<td>MATLAB</td>
</tr>
</tbody>
</table>
The TIFF image format is supported on all platforms by almost all word processors for importing images. JPEG is a lossy, highly compressed format that is supported on all platforms for image processing and for inclusion into HTML documents on the World Wide 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.

This table summarizes options that you can specify for `print`. The second column also shows which tutorial sections contain more detailed information. The sections listed are located under Printing and Exporting Figures with MATLAB.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-adobecset</td>
<td>PostScript only. Use PostScript default character set encoding. See Early PostScript 1 Printers.</td>
</tr>
<tr>
<td>-append</td>
<td>PostScript only. Append figure to existing PostScript file. See Settings That Are Driver Specific.</td>
</tr>
<tr>
<td>-cmyk</td>
<td>PostScript only. Print with CMYK colors instead of RGB. See Setting CMYK Color.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-ddriver</td>
<td>Printing only. Printer driver to use. See Drivers table.</td>
</tr>
<tr>
<td>-dformat</td>
<td>Exporting only. Graphics format to use. See Graphics Format Files table.</td>
</tr>
<tr>
<td>-dsetup</td>
<td>Display the Print Setup dialog.</td>
</tr>
<tr>
<td>-fhandle</td>
<td>Handle of figure to print. Note that you cannot specify both this option and the -swindowtitle option. See Which Figure Is Printed.</td>
</tr>
<tr>
<td>-loose</td>
<td>PostScript and Ghostscript only. Use loose bounding box for PostScript. See Producing Uncropped Figures.</td>
</tr>
<tr>
<td>-noui</td>
<td>Suppress printing of user interface controls. See Excluding User Interface Controls.</td>
</tr>
<tr>
<td>-opengl</td>
<td>Render using the OpenGL algorithm. Note that you cannot specify this method in conjunction with -zbuffer or -painters. See Selecting a Renderer.</td>
</tr>
<tr>
<td>-painters</td>
<td>Render using the Painter's algorithm. Note that you cannot specify this method in conjunction with -zbuffer or -opengl. See Selecting a Renderer.</td>
</tr>
<tr>
<td>-Pprinter</td>
<td>Specify name of printer to use. See Selecting Printer.</td>
</tr>
<tr>
<td>-rnumber</td>
<td>PostScript, JPEG, PNG, and Ghostscript 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. See Setting the Resolution.</td>
</tr>
</tbody>
</table>
Option | Description
--- | ---
-swINDOWTITLE | 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.
-v | Windows only. Display the Windows Print dialog box. The v stands for "verbose mode."
-zBUFFER | Render using the Z-buffer algorithm. Note that you cannot specify this method in conjunction with -OPENGL or -PAINTERS. See Selecting a Renderer.

**Paper Sizes**

MATLAB 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>11 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>Property Value</td>
<td>Size (Width by Height)</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------</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>

This section includes information about specific printing issues.

**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 MS 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 MS-Windows 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 the print command: they 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 onscreen 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

```
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 prevent MATLAB from using a bounding box that is 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.

**Notes on Printing Interpolated Shading with PostScript Drivers**

MATLAB can print surface objects (such as graphs created with `surf` or `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 time out 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 `-zbuffer` option. To obtain higher resolution (for example, to make text look better), use the `-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 PaperPositionMode to
auto, or by just setting the 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

    print -f handle

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.

    h = figure('IntegerHandle','off')
    print h -depson

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 filename to be passed in as a variable.

filename = 'mydata';
print('-f3', '-dpsc', filename);

(Because a filename 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 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 Options and the previous section 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 filenames stored in a cell array to create a series of graphs and prints each one with a different file name.

```matlab
fnames = {'file1', 'file2', 'file3'};
for k=1:length(fnames)
    surf(peaks)
    print('-dtiff','-r200',fnames{k})
end
```

**Tiff Preview**

The command

```matlab
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**

`orient`, `figure`
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 uimenus 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

**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**  
```matlab
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>Auto</td>
<td>Let MATLAB decide placement of plot on page</td>
</tr>
<tr>
<td></td>
<td>Use manual...</td>
<td>Specify position parameters for plot on page</td>
</tr>
<tr>
<td></td>
<td>Top, Left, Width,</td>
<td>Standard position parameters in current units</td>
</tr>
<tr>
<td></td>
<td>Height</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use defaults</td>
<td>Revert to default position</td>
</tr>
<tr>
<td></td>
<td>Fill page</td>
<td>Expand figure to fill printable area</td>
</tr>
<tr>
<td></td>
<td>Fix aspect ratio</td>
<td>Correct height/width ratio</td>
</tr>
<tr>
<td></td>
<td>Center</td>
<td>Center plot on printed page</td>
</tr>
<tr>
<td>Paper</td>
<td>Format</td>
<td>U.S. and ISO sheet size selector</td>
</tr>
<tr>
<td></td>
<td>Width, Height</td>
<td>Sheet size in current units</td>
</tr>
<tr>
<td>Units</td>
<td>Inches</td>
<td>Use inches as units for dimensions and positions</td>
</tr>
<tr>
<td></td>
<td>Centimeters</td>
<td>Use centimeters as units for dimensions and positions</td>
</tr>
<tr>
<td></td>
<td>Points</td>
<td>Use points as units for dimensions and positions</td>
</tr>
<tr>
<td>Orientation</td>
<td>Portrait</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>

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:
<table>
<thead>
<tr>
<th>Group</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>Line Width</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>Min Width</td>
<td>Smallest line width (in points) to use when printing; defaults to 0.5 point</td>
</tr>
<tr>
<td>Text</td>
<td>Font Name</td>
<td>Select a system font for all text on plot, or default to fonts currently used on the plot</td>
</tr>
<tr>
<td></td>
<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</td>
</tr>
<tr>
<td></td>
<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>
</tr>
<tr>
<td></td>
<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>
</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>
<tr>
<td>Group</td>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Color</td>
<td>Color</td>
<td>Print everything in color, matching colors on plot; select RGB (default) or CMYK color model for printing</td>
</tr>
<tr>
<td>Background Color</td>
<td>Same as figure</td>
<td>Print the figure’s background color as it is</td>
</tr>
<tr>
<td></td>
<td>Custom</td>
<td>Select a color name, or type a colorspec for the background; white (default) implies no background color, even on colored paper.</td>
</tr>
</tbody>
</table>

**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:

<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>
<tr>
<td>Group</td>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Group Option Description</td>
<td>Keep limits and ticks</td>
<td>Use the x- and y-axes ticks and limits shown on the plot when printing the previewed figure</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Renderer</td>
<td>Select a rendering algorithm for printing: painters, zbuffer, opengl, or auto (default)</td>
</tr>
<tr>
<td></td>
<td>Resolution</td>
<td>Select resolution to print at in dots per inch: 150, 300, 600, or auto (default), or type in any other positive value</td>
</tr>
<tr>
<td></td>
<td>Print UIControls</td>
<td>Print all visible UIControls in the figure (default), or uncheck to exclude them from being printed</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**

\[
B = \text{prod}(A) \\
B = \text{prod}(A,\text{dim})
\]

**Description**

\(B = \text{prod}(A)\) returns the products along different dimensions of an array.

If \(A\) is a vector, \(\text{prod}(A)\) returns the product of the elements.

If \(A\) is a matrix, \(\text{prod}(A)\) treats the columns of \(A\) as vectors, returning a row vector of the products of each column.

If \(A\) is a multidimensional array, \(\text{prod}(A)\) treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

\(B = \text{prod}(A,\text{dim})\) takes the products along the dimension of \(A\) specified by scalar \(\text{dim}\).

**Examples**

The magic square of order 3 is

\[
M = \text{magic}(3)
\]

\[
M = \\
\begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2 \\
\end{bmatrix}
\]

The product of the elements in each column is

\[
\text{prod}(M) = \\
\begin{bmatrix}
96 & 45 & 84 \\
\end{bmatrix}
\]

The product of the elements in each row can be obtained by:

\[
\text{prod}(M,2) = \\
\begin{bmatrix}
48 \\
\end{bmatrix}
\]
See Also

cumprod, diff, sum
Purpose
Profile execution time for function

GUI
As an alternative to the profile function, select Desktop > Profiler to open the Profiler.

Alternatives

Syntax
profile on
profile on -detail level
profile on -history
profile on -nohistory
profile on -timer clock
profile on -detail level -history -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 function in the M-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. Profile 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. To change options, stop profiling and then start or resume profiling with new options.

profile on starts the Profiler, clearing previously recorded profile statistics.

profile on -detail level starts the Profiler, clearing previously recorded profile statistics, and specifies the set of functions you want to profile. The level applies to subsequent uses of profile or the Profiler, until you change it. Allowable values for level are 2-2498.
• 'builtin'—Gathers information about M-functions, M-subfunctions, and MEX-functions, plus built-in functions, such as eig.

• 'mex'—Gathers information about M-functions, M-subfunctions, and MEX-functions. This is the default value.

profile on -history starts the Profiler, clearing previously recorded profile statistics, and records the exact sequence of function calls. The profile function records up to 10,000 function entry and exit events. For more than 10,000 events, profile continues to record other profile statistics, but not the sequence of calls. By default, the history option is not enabled.

profile on -nohistory starts the Profiler, clearing previously recorded profile statistics, and 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 accumulate.

profile on -timer clock starts the Profiler, clearing previously recorded profile statistics, and specifies the type of time to use. Allowable values for clock are

• 'cpu'—The Profiler uses compute time (the default).
• 'real'—The Profiler uses wall-clock time.

For example, cpu time for the pause function would be small, but real time would account for the actual time paused.

profile on -detail level -history -timer clock starts the Profiler using all of these specified 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 they appear in the structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProfilerStatus</td>
<td>'on' or 'off'</td>
</tr>
<tr>
<td>DetailLevel</td>
<td>'mmex' or 'builtin'</td>
</tr>
<tr>
<td>Timer</td>
<td>'cpu' or 'real'</td>
</tr>
<tr>
<td>HistoryTracking</td>
<td>'on' or 'off'</td>
</tr>
</tbody>
</table>

stats = profile('info') stops the Profiler and displays a structure containing the results. Use this function to access the data generated by profile. The table lists the fields in the order 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>
<tr>
<td>FunctionHistory</td>
<td>Array containing function call history</td>
</tr>
<tr>
<td>ClockPrecision</td>
<td>Precision of profile’s time measurement</td>
</tr>
<tr>
<td>ClockSpeed</td>
<td>Estimated clock speed of the CPU</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the profiler</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 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>
<tr>
<td>ExecutedLines</td>
<td>Array containing line-by-line details for the function being profiled.</td>
</tr>
<tr>
<td></td>
<td>Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix.</td>
</tr>
<tr>
<td></td>
<td>Column 2: Number of times the line was executed</td>
</tr>
<tr>
<td></td>
<td>Column 3: Total time spent on that line. Note: The sum of Column 3 entries does not necessarily add up to the function's TotalTime.</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-n array. The first row contains Boolean values, where 1 means entrance into a function and 0 means exit from a function. The second row identifies the function being entered or exited by its index in the `FunctionTable` field. This example reads the history data and displays it in the MATLAB Command Window.

```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
**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
### Purpose
Open built-in property page for control

### Syntax
- `h.propedit`
- `propedit(h)`

### Description
`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.

### Examples
Create a Microsoft Calendar control and display its property page:

```matlab
  cal = actxcontrol('mscal.calendar', [0 0 500 500]);
  cal.propedit
```

### See Also
- inspect
- get
**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**
- `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(figure_handle,...)` 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

\[
\psi(x) = \frac{d}{dx} \log(\Gamma(x)) = \frac{\frac{d\Gamma(x)}{dx}}{\Gamma(x)}
\]

Y = psi(k,X) evaluates the kth 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 \( \frac{\pi^2}{6} - 1 \).

```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 Alternatives

As an alternative to the publish function, use the File > Publish To menu items in the Editor/Debugger.

Syntax

\begin{verbatim}
publish('script')
publish('script','format')
publish('script', options)
publish('function', options)
\end{verbatim}

Description

\texttt{publish('script')} runs the M-file script named \texttt{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 \texttt{script.html} and is stored, along with other supporting output files, in an \texttt{html} subdirectory in \texttt{script's} directory.

\texttt{publish('script','format')} runs the M-file script named \texttt{script}, one cell at a time in the base workspace, and publishes the code, comments, and results to an output file using the specified \texttt{format}. Allowable values for \texttt{format} are \texttt{html} (the default), \texttt{xml}, \texttt{latex} for \LaTeX, \texttt{doc} for Microsoft Word documents, and \texttt{ppt} for Microsoft PowerPoint documents. The output file is named \texttt{script.format} and is stored, along with other supporting output files, in an \texttt{html} subdirectory in \texttt{script's} directory. The \texttt{doc} format requires the Microsoft Word application, and the \texttt{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 \texttt{false}. Because it is included as comments, it does not display in a Web browser. Use the \texttt{grabcode} function to extract the code from the HTML file.

\texttt{publish('script', options)} publishes using the structure \texttt{options}, which can contain any of the fields and corresponding value for each field as shown in Options for publish on page 2-2511. Create and save structures for the options you use regularly. For details about the values, see and Publishing Images preferences in the online documentation for MATLAB.
publish('function', options) publishes an M-file function using the structure options. The evalCode field must be set to false to publish a function. Publishing an M-file function 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 filename (used only when format is html, latex, or xml)</td>
</tr>
<tr>
<td>outputDir</td>
<td>'' (default, a subfolder named html), full pathname</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.</td>
</tr>
<tr>
<td>figureSnapMethod</td>
<td>'print' (default), 'getframe'</td>
</tr>
<tr>
<td>useNewFigure</td>
<td>true (default), false</td>
</tr>
<tr>
<td>maxHeight</td>
<td>[] (default), positive integer specifying number of pixels</td>
</tr>
<tr>
<td>maxWidth</td>
<td>[] (default), positive integer specifying number of pixels</td>
</tr>
<tr>
<td>showCode</td>
<td>true (default), false</td>
</tr>
<tr>
<td>evalCode</td>
<td>true (default), false</td>
</tr>
<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>stopOnError</td>
<td>true (default), false</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 before truncation of output</td>
</tr>
</tbody>
</table>
Examples

Publish to HTML Format

To publish the M-file script d:/mymfiles/sine_wave.m to HTML, run

    publish('d:/mymfiles/sine_wave.m', 'html')

MATLAB runs the file and saves the code, comments, and results to d:/mymfiles/html/sine_wave.html. Open that file in the Web browser to view the published document.

Publish with Options

This example defines the structure options_doc_nocode, publishes sine_wave.m using the defined options, and displays the resulting file. The resulting file is a Word document, d:/nocode_output/sine_wave.doc and includes results, but not MATLAB code.

    options_doc_nocode.format='doc'
    options_doc_nocode.outputDir='d:/nocode_output'
    options_doc_nocode.showCode=false
    publish('d:/mymfiles/sine_wave.m',options_doc_nocode)
    winopen('d:/nocode_output/sine_wave.doc')

Publish Function M-File (Save M-File as HTML)

This example defines the structure function_options, publishes the function d:/collatzplot.m, and displays the resulting file, an HTML document, d:/html/collatzplot.html.

    function_options.format='html'
    function_options.evalCode=false
    publish('d:/collatzplot.m',function_options)
    web('d:/html/collatzplot.html')

See Also

grabcode, notebook, web, winopen

MATLAB Desktop Tools and Development Environment documentation, specifically
- Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells
- Defining Cells


**Purpose**

Store character array in server

**Syntax**

**MATLAB Client**

```
h.PutCharArray('varname', 'workspace', 'string')
PutCharArray(h, 'varname', 'workspace', 'string')
invoke(h, 'PutCharArray', 'varname', 'workspace', 'string')
```

**Method Signature**

```
```

**Visual Basic Client**

```
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-

```
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. Retrieve the string with GetCharArray.

**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**

```vbnet
Dim Matlab As Object
Dim S As String
Matlab = CreateObject("matlab.application")
Matlab.PutCharArray("str", "base",
    "He jests at scars that never felt a wound.")
S = Matlab.GetCharArray("str", "base")
```

**See Also**

GetCharArray, PutWorkspaceData, GetWorkspaceData, Execute
**PutFullMatrix**

**Purpose**
Store matrix in server

**Syntax**

**MATLAB Client**

h.PutFullMatrix('varname', 'workspace', xreal, ximag)
PutFullMatrix(h, 'varname', 'workspace', xreal, ximag)
invoke(h, 'PutFullMatrix', 'varname', 'workspace', xreal, ximag)

**Method Signature**

PutFullMatrix([in] BSTR varname, [in] BSTR workspace,
[in] SAFEARRAY(double) xreal, [in] SAFEARRAY(double) ximag)

**Visual Basic Client**

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**

**Example 1 — Writing to the Base Workspace**

Assign a 5-by-5 real matrix to the variable M in the base workspace of the server, and then read it back with GetFullMatrix. The real and imaginary parts are passed in through separate arrays of doubles.

**MATLAB Client**

```matlab
h = actxserver('matlab.application');
h.PutFullMatrix('M', 'base', rand(5), zeros(5))
% One output returns real, use two for real and imag
xreal = h.GetFullMatrix('M', 'base', zeros(5), zeros(5))
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
```

**Visual Basic.net Client**

```vbnet
Dim MatLab As Object
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim ZReal(4, 4) As Double
Dim ZImag(4, 4) As Double
Dim i, j As Integer

For i = 0 To 4
    For j = 0 To 4
        XReal(i, j) = Rnd() * 6
        XImag(i, j) = 0
    Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("M", "base", XReal, XImag)
MatLab.GetFullMatrix("M", "base", ZReal, ZImag)
```
Example 2 — Writing to the Global Workspace

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                        
    X         2x3       96      double array (global complex)

Grand total is 6 elements using 96 bytes
```

Visual Basic.net Client

```vbs
Dim MatLab As Object
Dim XReal(1, 2) As Double
Dim XImag(1, 2) As Double
Dim result As String

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("M", "global", XReal, XImag)
result = Matlab.Execute("whos global")
MsgBox(result)
```
PutFullMatrix

See Also

GetFullMatrix, PutWorkspaceData, GetWorkspaceDataExecute
PutWorkspaceData

**Purpose**
Store data in server workspace

**Syntax**

**MATLAB Client**

```
% PutWorkspaceData
h.PutWorkspaceData('varname', 'workspace', data)
PutWorkspaceData(h, 'varname', 'workspace', data)
invoke(h, 'PutWorkspaceData', 'varname', 'workspace', data)
```

**Method Signature**

```
```

**Visual Basic Client**

```
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 data 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:

```
chArr = ['abc';'def';'ghk']
chArr =
abc
def
ghk

size(chArr)
an\_ =

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**

```vbs
Dim Matlab As Object
Dim data(6) As Double
MatLab = CreateObject("matlab.application")
For i = 0 To 6
    data(i) = i * 15
Next i
MatLab.PutWorkspaceData("A", "base", data)
```

**See Also**

GetWorkspaceData, PutFullMatrix, GetFullMatrix, PutCharArray, GetCharArrayExecute

See “Executing Commands in the MATLAB Server” for more examples.
**Purpose**
Identify current directory

**Graphical Interface**
As an alternative to the `pwd` function, use the “Current Directory Field” in the MATLAB desktop toolbar.

**Syntax**
```
pwd
s = pwd
```

**Description**
`pwd` displays the current working directory.

`s = pwd` returns the current directory to the variable `s`.

On Windows platforms, go directly to the current working directory using
```
winopen(pwd)
```

**See Also**
cd, dir, fileparts, mfilename, path, what, winopen

2-2523
**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 = 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 `afun` such that `afun(x,'notransp')` returns $A*x$ and `afun(x,'transp')` returns $A'*x$. See “Function Handles” in the MATLAB Programming documentation for more information.

- “Parameterizing Functions Called by Function Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

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

- `qmr(A,b,tol)` specifies the tolerance of the method. If `tol` is [], then `qmr` uses the default, $1e-6$.

- `qmr(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is [], then `qmr` uses the default, $\min(n,20)$.

- `qmr(A,b,tol,maxit,M)` and `qmr(A,b,tol,maxit,M1,M2)` use preconditioners $M$ or $M = M1*M2$ and effectively solve the system
inv(M)*A*x = inv(M)*b for x. If M is [] then qmr applies no preconditioner. M can be a function handle mfun such that mfun(x,'notransp') returns M\x and mfun(x,'transp') returns M'\x.

qmr(A,b,tol,maxit,M1,M2,x0) specifies the initial guess. If x0 is [], then qmr uses the default, an all zero vector.

[x,flag] = 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>qmr converged to the desired tolerance tol within maxit iterations.</td>
</tr>
<tr>
<td>1</td>
<td>qmr 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>The method stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during qmr became too small or too large to continue computing.</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,flag,relres] = qmr(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = qmr(A,b,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

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

**Examples**

**Example 1**

```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 = 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
```
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
end

When you enter

x1=run_qmr;

MATLAB displays the message

qmr converged at iteration 9 to a solution with relative residual 5.6e-009

Example 3

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.

[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.

[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 1.6571e-016 (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

\[
\begin{align*}
&[Q,R] = \text{qr}(A) \quad \text{(full and sparse matrices)} \\
&[Q,R] = \text{qr}(A,0) \quad \text{(full and sparse matrices)} \\
&[Q,R,E] = \text{qr}(A) \quad \text{(full matrices)} \\
&[Q,R,E] = \text{qr}(A,0) \quad \text{(full matrices)} \\
&X = \text{qr}(A,0) \quad \text{(full matrices)} \\
&R = \text{qr}(A) \quad \text{(sparse matrices)} \\
&[C,R] = \text{qr}(A,B) \quad \text{(sparse matrices)} \\
&R = \text{qr}(A,0) \quad \text{(sparse matrices)} \\
&[C,R] = \text{qr}(A,B,0) \quad \text{(sparse matrices)}
\end{align*}
\]

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 \(A = Q \times R\). 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 \(\text{qr}\) 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 \(A \times E = Q \times R\). 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 \(A(:, E) = Q \times R\). 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,

$$R' * R = A' * A$$

This approach avoids the loss of numerical information inherent in the computation of $A' * A$. 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 $C = Q' * B$ 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

$$\text{minimize}\|Ax - b\|$$

with two steps

$$[C, R] = qr(A, b)$$

$$x = R \backslash c$$

If $A$ is sparse but not square, MATLAB 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:

\[ [Q,R] = qr(A) \]

\[
Q =
\begin{bmatrix}
-0.0776 & -0.8331 & 0.5444 & 0.0605 \\
-0.3105 & -0.4512 & -0.7709 & 0.3251 \\
-0.5433 & -0.0694 & -0.0913 & -0.8317 \\
-0.7762 & 0.3124 & 0.3178 & 0.4461 \\
\end{bmatrix}
\]

\[
R =
\begin{bmatrix}
-12.8841 & -14.5916 & -16.2992 \\
0 & -1.0413 & -2.0826 \\
0 & 0 & 0.0000 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

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 tol is a tolerance used to decide if a diagonal element of R is negligible. If \([Q, R, E] = \text{qr}(A)\), then

\[
\text{tol} = \max(\text{size}(A)) \ast \text{eps} \ast \text{abs}(R(1,1))
\]

The solution \(x\) was computed using the factorization and the two steps

\[
\begin{align*}
y &= Q' \ast b; \\
x &= R \backslash y
\end{align*}
\]

The computed solution can be checked by forming \(A \ast x\). This equals \(b\) to within roundoff error, which indicates that even though the simultaneous equations \(A \ast x = b\) 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, \text{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, DORGQR</td>
<td>ZGEQRF, ZUNGQR</td>
</tr>
<tr>
<td>([Q, R] = \text{qr}(A))</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><code>R = qr(A)</code>&lt;br&gt;<code>R = qr(A,0)</code></td>
<td>SGEQRF</td>
<td>CGEQRF</td>
</tr>
<tr>
<td><code>[Q,R] = qr(A)</code>&lt;br&gt;<code>[Q,R] = qr(A,0)</code></td>
<td>SGEQRF, SORGQR</td>
<td>CGEQRF, CUNGQR</td>
</tr>
</tbody>
</table>

**See Also**

`lu`, `null`, `orth`, `qrdelete`, `qrinsert`, `qrupdate`

The arithmetic operators \\ and /

**References**

Purpose: Remove column or row from QR factorization

Syntax:

\[ [Q_1, R_1] = qrdelete(Q, R, j) \]
\[ [Q_1, R_1] = qrdelete(Q, R, j, 'col') \]
\[ [Q_1, R_1] = qrdelete(Q, R, j, 'row') \]

Description:

\[ [Q_1, R_1] = qrdelete(Q, R, j) \] returns the QR factorization of the matrix \( A_1 \), where \( A_1 \) is \( A \) with the column \( A(:,j) \) removed and \( [Q, R] = qr(A) \) is the QR factorization of \( A \).

\[ [Q_1, R_1] = qrdelete(Q, R, j, 'col') \] is the same as \( qrdelete(Q, R, j) \).

\[ [Q_1, R_1] = qrdelete(Q, R, j, 'row') \] returns the QR factorization of the matrix \( A_1 \), where \( A_1 \) is \( A \) with the row \( A(j,:) \) removed and \( [Q, R] = qr(A) \) is the QR factorization of \( A \).

Examples:

\[ A = magic(5); \]
\[ [Q, R] = qr(A); \]
\[ j = 3; \]
\[ [Q1, R1] = qrdelete(Q, R, j, 'row'); \]

\[ Q1 = \begin{bmatrix}
0.5274 & -0.5197 & -0.6697 & -0.0578 \\
0.7135 & 0.6911 & 0.0158 & 0.1142 \\
0.3102 & -0.1982 & 0.4675 & -0.8037 \\
0.3413 & -0.4616 & 0.5768 & 0.5811 \\
\end{bmatrix} \]

\[ R1 = \begin{bmatrix}
0 & -19.7045 & -10.9891 & 0.4318 & -1.4873 \\
0 & 0 & 22.7444 & 5.8357 & -3.1977 \\
0 & 0 & 0 & -14.5784 & 3.7796 \\
\end{bmatrix} \]

returns a valid QR factorization, although possibly different from

\[ A2 = A; \]
\[ A2(j,:) = []; \]
\[ [Q2, R2] = qr(A2) \]
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**

\[
\begin{align*}
[Q1,R1] &= \text{qrinsert}(Q,R,j,x) \\
[Q1,R1] &= \text{qrinsert}(Q,R,j,x,'\text{col}') \\
[Q1,R1] &= \text{qrinsert}(Q,R,j,x,'\text{row}')
\end{align*}
\]

**Description**

\[
[Q1,R1] = \text{qrinsert}(Q,R,j,x) \quad \text{returns the QR factorization of the matrix } A1, \text{where } A1 = Q*R \text{ with the column } x \text{ inserted before } A(:,j). \text{ If } A \text{ has } n \text{ columns and } j = n+1, \text{ then } x \text{ is inserted after the last column of } A.
\]

\[
[Q1,R1] = \text{qrinsert}(Q,R,j,x,'\text{col}') \quad \text{is the same as } \text{qrinsert}(Q,R,j,x).
\]

\[
[Q1,R1] = \text{qrinsert}(Q,R,j,x,'\text{row}') \quad \text{returns the QR factorization of the matrix } A1, \text{where } A1 = Q*R \text{ with an extra row, } x, \text{ inserted before } A(j,:).
\]

**Examples**

\[
A = \text{magic}(5);
\]

\[
[Q,R] = \text{qr}(A);
\]

\[
j = 3;
\]

\[
x = 1:5;
\]

\[
[Q1,R1] = \text{qrinsert}(Q,R,j,x,'\text{row}')
\]

\[
Q1 =
\begin{bmatrix}
0.5231 & 0.5039 & -0.6750 & 0.1205 & 0.0411 & 0.0225 \\
0.7078 & -0.6966 & 0.0190 & -0.0788 & 0.0833 & -0.0150 \\
0.0308 & 0.0592 & 0.0656 & 0.1169 & 0.1527 & -0.9769 \\
0.1231 & 0.1363 & 0.3542 & 0.6222 & 0.6398 & 0.2104 \\
0.3077 & 0.1902 & 0.4100 & 0.4161 & -0.7264 & -0.0150 \\
0.3385 & 0.4500 & 0.4961 & -0.6366 & 0.1761 & 0.0225
\end{bmatrix}
\]

\[
R1 =
\begin{bmatrix}
0 & 19.9292 & 12.4403 & 2.1340 & 4.3271 \\
0 & 0 & 24.4514 & 11.8132 & 3.9931 \\
0 & 0 & 0 & 20.2382 & 10.3392
\end{bmatrix}
\]
returns a valid QR factorization, although possibly different from

\[ A2 = [A(1:j-1,:); x; A(j:end,:)]; \]
\[ [Q2,R2] = qr(A2) \]

\[
\begin{array}{cccccc}
-0.5231 & 0.5039 & 0.6750 & -0.1205 & 0.0411 & 0.0225 \\
-0.7078 & -0.6966 & -0.0190 & 0.0788 & 0.0833 & -0.0150 \\
-0.0308 & 0.0592 & -0.0656 & -0.1169 & 0.1527 & -0.9769 \\
-0.1231 & 0.1363 & -0.3542 & -0.6222 & 0.6398 & 0.2104 \\
-0.3077 & 0.1902 & -0.4100 & -0.4161 & -0.7264 & -0.0150 \\
-0.3385 & 0.4500 & -0.4961 & 0.6366 & 0.1761 & 0.0225
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 19.9292 & 12.4403 & 2.1340 & 4.3271 \\
0 & 0 & -24.4514 & -11.8132 & -3.9931 \\
0 & 0 & 0 & -20.2382 & -10.3392 \\
0 & 0 & 0 & 0 & 16.1948 \\
0 & 0 & 0 & 0 & 0
\end{array}
\]

**Algorithm**

The `qrinsert` function inserts the values of `x` into the `j`th 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 `j`th 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

\[
\text{mu} = \sqrt{\text{eps}}
\]

\[
\text{mu} = \\
1.4901e-08
\]

\[
A = [\text{ones}(1,4); \text{mu*eye}(4)];
\]

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 $\sqrt{\varepsilon}$.

Consider the update vectors

\[ u = [-1 \ 0 \ 0 \ 0 \ 0]'; \ v = \text{ones}(4,1); \]

Instead of computing the rather trivial QR factorization of this rank one update to $A$ from scratch with

\[ [QT,RT] = \text{qr}(A + u*v') \]

\[ QT = \]

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 \\
-1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 \\
\end{bmatrix}
\]

\[ RT = \]

\[
1.0\cdot10^{-007} * \\
\begin{bmatrix}
-0.1490 & 0 & 0 & 0 & 0 \\
0 & -0.1490 & 0 & 0 & 0 \\
0 & 0 & -0.1490 & 0 & 0 \\
0 & 0 & 0 & -0.1490 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

we may use \texttt{qrupdate}.

\[ [Q1,R1] = \text{qrupdate}(Q,R,u,v) \]

\[ Q1 = \]

\[
\begin{bmatrix}
-0.0000 & -0.0000 & -0.0000 & -0.0000 & 1.0000 \\
1.0000 & -0.0000 & -0.0000 & -0.0000 & 0.0000 \\
\end{bmatrix}
\]
$$\begin{bmatrix}
0.0000 & 1.0000 & -0.0000 & -0.0000 & 0.0000 \\
0.0000 & 0.0000 & 1.0000 & -0.0000 & 0.0000 \\
-0.0000 & -0.0000 & -0.0000 & 1.0000 & 0.0000 
\end{bmatrix}$$

\[ R1 = \]

$$\begin{bmatrix}
1.0e-007 & * \\
0.1490 & 0.0000 & 0.0000 & 0.0000 \\
0 & 0.1490 & 0.0000 & 0.0000 \\
0 & 0 & 0.1490 & 0.0000 \\
0 & 0 & 0 & 0.1490 \\
0 & 0 & 0 & 0 
\end{bmatrix}$$

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
q = quad(fun,a,b)
q = quad(fun,a,b,tol)
q = quad(fun,a,b,tol,trace)
[q,fcnt] = quad(...)

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 \( x \).

“Parameterizing Functions Called by Function 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(...) returns the number of function evaluations.

The function quadl may be more efficient with high accuracies and smooth integrands.
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:

```matlab
Q = quad(@myfun,0,2)
```

\[Q = -0.4605\]

Alternatively, you can pass the integrand to `quad` as an anonymous function handle `F`:

```matlab
F = @(x)1./(x.^3-2*x-5);
Q = 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**

dblquad, quadl, quadv, trapz, triplequad, function_handle (@), “Anonymous Functions”

**References**

**Purpose**  
Numerically evaluate integral, adaptive Lobatto quadrature

**Syntax**
```matlab
q = quadl(fun,a,b)  
q = quadl(fun,a,b,tol)  
quadl(fun,a,b,tol,trace)  
[q,fcnt] = quadl(...)```

**Description**
$q = \text{quadl}(\text{fun},a,b)$ approximates the integral of function $\text{fun}$ from $a$ to $b$, to within an error of $10^{-6}$ using recursive adaptive Lobatto quadrature. $\text{fun}$ is a function handle. See “Function Handles” in the MATLAB Programming documentation for more information. $\text{fun}$ accepts a vector $x$ and returns a vector $y$, the function $\text{fun}$ evaluated at each element of $x$. Limits $a$ and $b$ must be finite.

“Parameterizing Functions Called by Function Functions”, in the MATLAB Mathematics documentation, explains how to provide additional parameters to the function $\text{fun}$, if necessary.

$q = \text{quadl}(\text{fun},a,b,tol)$ uses an absolute error tolerance of $\text{tol}$ instead of the default, which is $1.0e^{-6}$. Larger values of $\text{tol}$ result in fewer function evaluations and faster computation, but less accurate results.

$q = \text{quadl}(\text{fun},a,b,tol,\text{trace})$ with non-zero $\text{trace}$ shows the values of $[\text{fcnt} \ a \ b-a \ q]$ during the recursion.

$[q,\text{fcnt}] = \text{quadl}(\ldots)$ returns the number of function evaluations.

Use array operators $\cdot \ast$, $\cdot /$ and $\cdot ^$ in the definition of $\text{fun}$ so that it can be evaluated with a vector argument.

The function quad may be more efficient with low accuracies or nonsmooth integrands.

**Examples**
Pass M-file function handle @myfun to quadl:

```matlab
Q = quadl(@myfun,0,2);
```

where the M-file myfun.m is
function y = myfun(x)
    y = 1./(x.^3-2*x-5);

Pass anonymous function handle F to quadl:

    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.

**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**

dblquad, quad, triplequad, function_handle (@), “Anonymous Functions”

**References**

### Purpose
Vectorized quadrature

### Syntax

- \( Q = \text{quadv}(\text{fun},a,b) \)
- \( Q = \text{quadv}(\text{fun},a,b,\text{tol}) \)
- \( Q = \text{quadv}(\text{fun},a,b,\text{tol},\text{trace}) \)
- \([Q,\text{fcnt}] = \text{quadv}(\ldots)\)

### Description

- \( Q = \text{quadv}(\text{fun},a,b) \) approximates the integral of the complex array-valued function \( \text{fun} \) from \( a \) to \( b \) to within an error of \( 1.\times10^{-6} \) using recursive adaptive Simpson quadrature. \( \text{fun} \) 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.

- “Parameterizing Functions Called by Function Functions”, in the MATLAB Mathematics documentation, explains how to provide addition parameters to the function \( \text{fun} \), if necessary.

- \( Q = \text{quadv}(\text{fun},a,b,\text{tol}) \) uses the absolute error tolerance \( \text{tol} \) for all the integrals instead of the default, which is \( 1.\times10^{-6} \).

- \( Q = \text{quadv}(\text{fun},a,b,\text{tol},\text{trace}) \) with non-zero \( \text{trace} \) shows the values of \([\text{fcnt} \ a \ b-a \ Q(1)]\) during the recursion.

- \([Q,\text{fcnt}] = \text{quadv}(\ldots)\) returns the number of function evaluations.

### Note
The same tolerance is used for all components, so the results obtained with \text{quadv} are usually not the same as those obtained with \text{quad} on the individual components.

### Example
For the parameterized array-valued function \( \text{myarrayfun} \), defined by

```matlab
function Y = myarrayfun(x,n)
    Y = 1./(1:n+x);
```
the following command integrates \texttt{myarrayfun}, for the parameter value \( n = 10 \) between \( a = 0 \) and \( b = 1 \):

\[
Qv = \text{quadv}(@(x)\text{myarrayfun}(x,10),0,1);
\]

The resulting array \( Qv \) has 10 elements estimating \( Q(k) = \log\left(\frac{(k+1)}{k}\right) \), for \( k = 1:10 \).

The entries in \( Qv \) are slightly different than if you compute the integrals using \texttt{quad} in a loop:

\[
\text{for } k = 1:10 \\
\quad Qs(k) = \text{quadv}(@(x)\text{myscalarfun}(x,k),0,1);
\text{end}
\]

where \texttt{myscalarfun} is:

\[
\text{function } y = \text{myscalarfun}(x,k) \\
\quad y = 1.0/(k+x);
\]

\textbf{See Also} \quad \texttt{quad}, \texttt{quadl}, \texttt{dblquad}, \texttt{triplequad}, \texttt{function_handle (@)}
**Purpose**
Create and open question dialog box

**Syntax**

```matlab
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')
```

**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, `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'`. 
questdlg

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'.

In all cases where 'default' is specified, if 'default' is not set to one of the button names, pressing the Enter key displays a warning and the dialog remains open.

See Also
dialog, errordlg, helpdlg, inputdlg, listdlg, msgbox, warndlg
figure, textwrap, uiwait, uiresume

“Predefined Dialog Boxes” on page 1-103 for related functions
**Purpose**
Terminate MATLAB

**GUI Alternatives**
As an alternative to the quit function, use the Close box or select **File > Exit MATLAB** in the MATLAB desktop.

**Syntax**
```matlab
quite 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 in `finish.m`, use `uiwait`, `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.

- `finishsav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.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`
**Purpose**
Termiate MATLAB server

**Syntax**

### MATLAB Client

- `h.Quit`
- `Quit(h)`
- `invoke(h, 'Quit')`

### Method Signature

`void Quit(void)`

### Visual Basic Client

`Quit`

**Description**

`Quit` terminates the MATLAB server session to which handle `h` is attached.

**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.
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 \(x(1), y(1)\).

\(\text{quiver}(x, y, u, v)\) 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 \texttt{meshgrid} to generate matrices from vectors:
\[ [x,y] = \text{meshgrid}(x,y); \]
\[ \text{quiver}(x,y,u,v) \]

In this case, the following must be true:
\[ \text{length}(x) = n \text{ and } \text{length}(y) = m, \text{ where } [m,n] = \text{size}(u) = \text{size}(v). \]
The vector \( x \) corresponds to the columns of \( u \) and \( v \), and vector \( y \) corresponds to the rows of \( u \) and \( v \).

\text{quiver}(u,v) \] draws vectors specified by \( u \) and \( v \) at equally spaced points in the \( x-y \) plane.

\text{quiver}(...,\text{scale}) \] automatically scales the arrows to fit within the grid and then stretches them by the factor \( \text{scale} \). \( \text{scale} = 2 \) doubles their relative length, and \( \text{scale} = 0.5 \) halves the length. Use \( \text{scale} = 0 \) to plot the velocity vectors without automatic scaling. You can also tune the length of arrows after they have been drawn by choosing the \textbf{Plot Edit} tool, selecting the \textit{quivergroup} object, opening the Property Editor, and adjusting the \textbf{Length} slider.

\text{quiver}(...,\text{LineSpec}) \] specifies line style, marker symbol, and color using any valid \text{LineSpec}. \text{quiver} draws the markers at the origin of the vectors.

\text{quiver}(...,\text{LineSpec},'\text{filled}') \] fills markers specified by \text{LineSpec}.

\text{quiver(axes\_handle,...)} \] plots into the axes with the handle \text{axes\_handle} instead of into the current axes (\text{gca}).

\( h = \text{quiver}(...) \) returns the handle to the \textit{quivergroup} object.

\textbf{Backward-Compatible Version}

\( h\text{lines} = \text{quiver('v6',...)} \) returns the handles of line objects instead of \textit{quivergroup} objects for compatibility with MATLAB 6.5 and earlier.

\textbf{Examples}

\textbf{Showing the Gradient with Quiver Plots}

Plot the gradient field of the function \( z = xe^{-(x^2-y^2)}. \)
[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 1-88 for related functions
Two-Dimensional Quiver Plots for more examples
Quivergroup Properties for property descriptions
**Purpose**

3-D 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**

- `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 \((x,y,z)\).

`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.

`quiver3(z,u,v,w)` plots the vectors at the equally spaced surface points specified by matrix \(z\). `quiver3` automatically scales the vectors based on the distance between them to prevent them from overlapping.

`quiver3(...,scale)` automatically scales the vectors to prevent them from overlapping, and then multiplies them by \(scale\). \(scale = 2\) doubles their relative length, and \(scale = 0.5\) halves them. Use \(scale = 0\) to plot the vectors without the automatic scaling.

`quiver3(...,LineSpec)` specifies line type and color using any valid `LineSpec`.

---

2-2557
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 \( z = xe^{(-x^2-y^2)} \).

\[
\begin{align*}
[X,Y] &= \text{meshgrid}([-2:0.25:2,-1:0.2:1]); \\
Z &= X.* \exp(-X.^2 - Y.^2); \\
[U,V,W] &= \text{surfnorm}(X,Y,Z); \\
quiver3(X,Y,Z,U,V,W,0.5); \\
\text{hold on} \\
\text{surf}(X,Y,Z); \\
\text{colormap hsv} \\
\text{view}(-35,45) \\
\text{axis}([-2 2 -1 1 -.6 .6]) \\
\text{hold off}
\end{align*}
\]
See Also

axis, contour, LineSpec, plot, plot3, quiver, surfnorm, view
“Direction and Velocity Plots” on page 1-88 for related functions
Three-Dimensional Quiver Plots for more examples
### 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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoScale</td>
<td>{on}</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoScaleFactor</td>
<td>scalar (default = 0.9)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>BeingDeleted</td>
<td>on</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
Quivergroup Properties

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.

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,

\[
\text{area}(y, 'CreateFcn', @CallbackFcn)
\]

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

_Label used by plot legends._ The legend function, the figure’s active legend, and the plot browser use this text when displaying labels for this object.

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.

**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.

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>
</tbody>
</table>
**Specifier String**

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
Marker Specifier | Description
---|---
^ | Upward-pointing triangle
v | Downward-pointing triangle
> | Right-pointing triangle
< | Left-pointing triangle
p | Five-pointed star (pentagram)
h | Six-pointed star (hexagram)
none | No marker (default)

**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.

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, 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

*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.
Quivergroup Properties

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 `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)

*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');```
Quivergroup 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.

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)
**Quivergroup Properties**

*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 \( \text{UData}(1), \text{VData}(1), \text{WData}(1) \).

**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.

### 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.
Quivergroup Properties

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  
\[
[AA, BB, Q, Z] = \text{qz}(A, B)
\]
\[
[AA, BB, Q, Z, V, W] = \text{qz}(A, B)
\]
\[
\text{qz}(A, B, \text{flag})
\]

Description  
The \text{qz} function gives access to intermediate results in the computation of generalized eigenvalues.

\[
[AA, BB, Q, Z] = \text{qz}(A, B)
\]  
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.

\[
[AA, BB, Q, Z, V, W] = \text{qz}(A, B)
\]  
also produces matrices \( V \) and \( W \) whose columns are generalized eigenvectors.

\[
\text{qz}(A, B, \text{flag})
\]  
for real matrices \( A \) and \( B \), produces one of two decompositions depending on the value of \( \text{flag} \):

'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.
\[ \lambda = \alpha ./ \beta \]

If \( AA \) is triangular, the diagonal elements of \( AA \) and \( BB \),

\[
\begin{align*}
\alpha &= \text{diag}(AA) \\
\beta &= \text{diag}(BB)
\end{align*}
\]

are the generalized eigenvalues that satisfy

\[
\begin{align*}
A*V*\text{diag}(\beta) &= B*V*\text{diag}(\alpha) \\
\text{diag}(\beta)*W'*A &= \text{diag}(\alpha)*W'*B
\end{align*}
\]

The eigenvalues produced by

\[
\text{lambda} = \text{eig}(A,B)
\]

are the element-wise ratios of \( \alpha \) and \( \beta \).

\[
\text{lambda} = \alpha ./ \beta
\]

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 \( B \), \textit{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**

\textit{eig}
References

Purpose
Uniformly distributed pseudorandom numbers

Syntax
Y = rand
Y = rand(n)
Y = rand(m,n)
Y = rand([m n])
Y = rand(m,n,p,...)
Y = rand([m n p...])
Y = rand(size(A))
rnd(method,s)
s = rand(method)

Description
Y = rand returns a pseudorandom, scalar value drawn from a uniform
distribution on the unit interval.

Y = rand(n) returns an n-by-n matrix of values derived as described
above.

Y = rand(m,n) or Y = rand([m n]) returns an m-by-n matrix of the
same.

Y = rand(m,n,p,...) or Y = rand([m n p...]) generates an
m-by-n-by-p-by-... array of the same.

Note The size inputs m, n, p, ... should be nonnegative integers.
Negative integers are treated as 0.

Y = rand(size(A)) returns an array that is the same size as A.

rand(method,s) causes rand to use the generator determined by
method, and initializes the state of that generator using the value of s.

The value of s is dependent upon which method is selected. If method
is set to 'state' or 'twister', then s must be either a scalar integer
value from 0 to 2^32-1 or the output of rand(method). If method is set
to 'seed', then s must be either a scalar integer value from 0 to 2^31-2
or the output of rand(method).
The `rand` and `randn` generators each maintain their own internal state information. Initializing the state of one has no effect on the other.

Input argument `method` can be any of the strings shown in the table below:

<table>
<thead>
<tr>
<th>method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'twister'</td>
<td>Use the Mersenne Twister algorithm by Nishimura and Matsumoto (the default in MATLAB Versions 7.4 and later). This method generates double-precision values in the closed interval ([2^\text{(-53)}, 1-2^\text{(-53)}]), with a period of ((2^{19937}-1)/2).</td>
</tr>
<tr>
<td>'state'</td>
<td>Use a modified version of Marsaglia's <em>subtract with borrow</em> algorithm (the default in MATLAB versions 5 through 7.3). This method can generate all the double-precision values in the closed interval ([2^\text{(-53)}, 1-2^\text{(-53)}]). It theoretically can generate over (2^{1492}) values before repeating itself.</td>
</tr>
<tr>
<td>'seed'</td>
<td>Use a multiplicative congruential algorithm (the default in MATLAB version 4). This method generates double-precision values in the closed interval ([1/(2^{31}-1), 1-1/(2^{31}-1)]), with a period of (2^{31}-2).</td>
</tr>
</tbody>
</table>

For a full description of the Mersenne twister algorithm, see [http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html](http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html)

`s = rand(method)` returns in `s` the current internal state of the generator selected by `method`. It does not change the generator being used.

**Remarks**

The sequence of numbers produced by `rand` is determined by the internal state of the generator. Setting the generator to the same fixed state enables you to repeat computations. Setting the generator to different states leads to unique computations. It does not, however, improve statistical properties.
Because MATLAB resets the rand state at startup, rand generates the same sequence of numbers in each session unless you change the value of the state input.

**Examples**

**Example 1**

Make a random choice between two equally probable alternatives:

```matlab
if rand < .5
    'heads'
else
    'tails'
end
```

**Example 2**

Generate a 3-by-4 pseudorandom matrix:

```matlab
R = rand(3,4)
R =
```

```matlab
0.8147 0.9134 0.2785 0.9649
0.9058 0.6324 0.5469 0.1576
0.1270 0.0975 0.9575 0.9706
```

**Example 3**

Set rand to its default initial state:

```matlab
rand('twister', 5489);
```

Initialize rand to a different state each time:

```matlab
rand('twister', sum(100*clock));
```

Save the current state, generate 100 values, reset the state, and repeat the sequence:

```matlab
s = rand('twister');
u1 = rand(100);
rand('twister',s);
```
u2 = rand(100); % contains exactly the same values as u1

**Example 4**

Generate uniform integers on the set 1:n:

```matlab
n = 75;
f = ceil(n.*rand(100,1));
```

```matlab
f(1:10)
ans =
72
37
61
11
32
69
60
72
50
3
```

**Example 5**

Generate a uniform distribution of random numbers on a specified interval [a,b]. To do this, multiply the output of rand by (b-a), then add a. For example, to generate a 5-by-5 array of uniformly distributed random numbers on the interval [10,50],

```matlab
a = 10; b = 50;
x = a + (b-a) * rand(5)
x =
19.1591  49.8454  10.1854  25.9913  17.2739
46.5335  13.1270  40.9964  20.3948  20.5521
16.0951  27.7071  42.6921  42.0027  15.8216
43.0327  14.2661  44.7478  27.2566  15.4427
31.5337  48.4759  13.3774  46.4259  44.7717
```
References


See Also

randn, randperm, sprand, sprandn
Purpose

Normally distributed random numbers

Syntax

\[
Y = \text{randn}
\]
\[
Y = \text{randn}(n)
\]
\[
Y = \text{randn}(m, n)
\]
\[
Y = \text{randn}([m \ n])
\]
\[
Y = \text{randn}(m, n, p, ...)
\]
\[
Y = \text{randn}([m \ n \ p ...])
\]
\[
Y = \text{randn}(	ext{size}(A))
\]
\[
\text{randn(method,s)}
\]
\[
s = \text{randn(method)}
\]

Description

\[ Y = \text{randn} \] returns a pseudorandom, scalar value drawn from a normal distribution with mean 0 and standard deviation 1.

\[ Y = \text{randn}(n) \] returns an \(n\)-by-\(n\) matrix of values derived as described above.

\[ Y = \text{randn}(m, n) \text{ or } Y = \text{randn}([m \ n]) \] returns an \(m\)-by-\(n\) matrix of the same.

\[ Y = \text{randn}(m, n, p, ...) \text{ or } Y = \text{randn}([m \ n \ p ...]) \] generates an \(m\)-by-\(n\)-by-\(p\)-by-\(...\) array of the same.

Note

The size inputs \(m, n, p, ...\) should be nonnegative integers. Negative integers are treated as 0.

\[ Y = \text{randn}(	ext{size}(A)) \] returns an array that is the same size as \(A\).

\text{randn(method,s)} \] causes \text{randn} to use the generator determined by \text{method}, and initializes the state of that generator using the value of \text{s}.

The value of \text{s} is dependent upon which \text{method} is selected. If \text{method} is set to 'state', then \text{s} must be either a scalar integer value from 0 to \(2^{32}-1\) or the output of \text{rand(method)}. If \text{method} is set to 'seed', then \text{s} must be either a scalar integer value from 0 to \(2^{31}-2\) or the
output of \texttt{rand} (\texttt{method}). To set the generator to its default initial state, set \(s\) equal to zero.

The \texttt{randn} and \texttt{rand} generators each maintain their own internal state information. Initializing the state of one has no effect on the other.

Input argument \texttt{method} can be either of the strings shown in the table below:

<table>
<thead>
<tr>
<th>\textbf{method}</th>
<th>\textbf{Description}</th>
</tr>
</thead>
<tbody>
<tr>
<td>'state'</td>
<td>Use Marsaglia’s ziggurat algorithm (the default in MATLAB versions 5 and later). The period is approximately (2^64).</td>
</tr>
<tr>
<td>'seed'</td>
<td>Use the polar algorithm (the default in MATLAB version 4). The period is approximately ((2^{31}-1) \times (\pi/8)).</td>
</tr>
</tbody>
</table>

\(s = \texttt{randn}(\texttt{method})\) returns in \(s\) the current internal state of the generator selected by \texttt{method}. It does not change the generator being used.

**Examples**

**Example 1**

\(R = \texttt{randn}(3,4)\) might produce

\[
R =
\begin{bmatrix}
1.1650 & 0.3516 & 0.0591 & 0.8717 \\
0.6268 & -0.6965 & 1.7971 & -1.4462 \\
0.0751 & 1.6961 & 0.2641 & -0.7012
\end{bmatrix}
\]

For a histogram of the \texttt{randn} distribution, see \texttt{hist}.

**Example 2**

Set \texttt{randn} to its default initial state:

\[
\texttt{randn('state', 0);}
\]

Initialize \texttt{randn} to a different state each time:
randn('state', sum(100*clock));

Save the current state, generate 100 values, reset the state, and repeat the sequence:

\[
\begin{align*}
  s &= \text{randn('state')} ; \\
  u1 &= \text{randn(100)} ; \\
  \text{randn('state',s)} ; \\
  u2 &= \text{randn(100)} ; \quad \text{% Contains exactly the same values as } u1.
\end{align*}
\]

**Example 3**

Generate a random distribution with a specific mean and variance \( \sigma^2 \). To do this, multiply the output of `randn` by the standard deviation \( \sigma \), and then add the desired mean. For example, to generate a 5-by-5 array of random numbers with a mean of .6 that are distributed with a variance of 0.1,

\[
x = .6 + \sqrt{0.1} \cdot \text{randn}(5)
\]

\[
x = \\
0.8713 & 0.4735 & 0.8114 & 0.0927 & 0.7672 \\
0.9966 & 0.8182 & 0.9766 & 0.6814 & 0.6694 \\
0.0960 & 0.8579 & 0.2197 & 0.2659 & 0.3085 \\
0.1443 & 0.8251 & 0.5937 & 1.0475 & -0.0864 \\
0.7806 & 1.0080 & 0.5504 & 0.3454 & 0.5813
\]

**References**


See Also

rand, randperm, sprand, sprandn
**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 `rand`'s state.

**Examples**
\( \text{randperm}(6) \) might be the vector
\[
[ 3 \ 2 \ 6 \ 4 \ 1 \ 5 ]
\]
or it might be some other permutation of \( 1:6 \).

**See Also**
`permute`
Purpose

Rank of matrix

Syntax

\[ k = \text{rank}(A) \]
\[ k = \text{rank}(A,\text{tol}) \]

Description

The \text{rank} function provides an estimate of the number of linearly independent rows or columns of a full matrix.

\[ k = \text{rank}(A) \]
returns the number of singular values of \( A \) that are larger than the default tolerance, \( \max(\text{size}(A))*\text{eps}(\text{norm}(A)) \).

\[ k = \text{rank}(A,\text{tol}) \]
returns the number of singular values of \( A \) that are larger than \( \text{tol} \).

Remark

Use \text{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 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

\[
\begin{align*}
    s &= \text{svd}(A); \\
    \text{tol} &= \max(\text{size}(A))*\text{eps}(\max(s)); \\
    r &= \text{sum}(s > \text{tol});
\end{align*}
\]

See Also

\text{sprank}

References

**Purpose**

Rational fraction approximation

**Syntax**

```
[N,D] = rat(X)
[N,D] = rat(X,tol)
rat(X)
S = rats(X,strlen)
S = rats(X)
```

**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 *rat* function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The *rats* function calls *rat*, and returns strings.

```
[N,D] = rat(X) returns arrays N and D so that N./D approximates X to within the default tolerance, 1.e-6*norm(X(:),1).
[N,D] = rat(X,tol) returns N./D approximating X to within tol.
rat(X), with no output arguments, simply displays the continued fraction.
```

```
S = rats(X,strlen) returns a string containing simple rational approximations to the elements of X. Asterisks are used for elements that cannot be printed in the allotted space, but are not negligible compared to the other elements in X. *strlen* is the length of each string element returned by the *rats* function. The default is *strlen = 13*, which allows 6 elements in 78 spaces.
```

```
S = rats(X) returns the same results as those printed by MATLAB with format *rat*.
```

**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 = 
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 `rat(s)` produces

```
1 + 1/(-4 + 1/(-6 + 1/(-3 + 1/(-5))))
```

And the statement

```
[n,d] = rat(s)
```

produces

```
n = 319, 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 \( 2^{52} \):

```
141484755040.568804503599627370496
```
However, this is not a simple rational number. The value printed for \( \pi \) with format \( \text{rat} \), or with \( \text{rats}(\pi) \), is

\[
355/113
\]

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 \( \text{rat}(\pi) \) produces

\[
3 + 1/(7 + 1/(16))
\]

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 \( \text{rat}(X) \) function approximates each element of \( X \) by a continued fraction of the form

\[
\frac{n}{d} = d_1 + \frac{1}{d_2 + \frac{1}{\left( d_3 + \ldots + \frac{1}{d_k} \right)}}
\]

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 \( 2.68 \times \left(0.173\right)^k \), so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.
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 1-98 for related functions
Purpose
Matrix reciprocal condition number estimate

Syntax
c = rcond(A)

Description
c = 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, rcond(A) is near 1.0. If A is badly conditioned, rcond(A) is near 0.0. Compared to cond, rcond is a more efficient, but less reliable, method of estimating the condition of a matrix.

Algorithm
For full matrices A, 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, DGECON</td>
<td>ZLANGE, ZGETRF, ZGECON</td>
</tr>
<tr>
<td>A single</td>
<td>SLANGE, SGETRF, SGECON</td>
<td>CLANGE, CGETRF, CGECON</td>
</tr>
</tbody>
</table>

See Also
cond, condest, norm, normest, rank, svd

References
Purpose

Read data asynchronously from device

Syntax

```plaintext
readasync(obj)
readasync(obj,size)
```

Arguments

- `obj` A serial port object.
- `size` The number of bytes to read from the device.

Description

`readasync(obj)` initiates an asynchronous read operation.

`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, 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)

See Also

Functions
fopen, stopasync

Properties
BytesAvailable, BytesAvailableFcn, ReadAsyncMode, Status, TransferStatus
## Purpose
Real part of complex number

## Syntax
\[ X = \text{real}(Z) \]

## Description
\[ X = \text{real}(Z) \] returns the real part of the elements of the complex array \( Z \).

## Examples
\[ \text{real}(2+3i) \text{ is 2.} \]

## See Also
abs, angle, conj, i, j, imag
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 = realmax

Description
n = realmax returns the largest floating-point number representable on your computer. Anything larger overflows.

realmax('double') is the same as realmax with no arguments.

realmax('single') is the largest single precision floating point number representable on your computer. Anything larger overflows to single(Inf).

Examples
realmax is one bit less than $2^{1024}$ or about 1.7977e+308.

Algorithm
The realmax function is equivalent to $\text{pow2}(2 - \text{eps}, \text{maxexp})$, where maxexp is the largest possible floating-point exponent.

Execute type realmax to see maxexp for various computers.

See Also
eps, realmin, intmax
Purpose
Smallest positive 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 $\text{pow}(2, \text{minexp})$ where \text{minexp} is the smallest possible floating-point exponent.

Execute type realmin to see \text{minexp} for various computers.

See Also
eps, realmax, intmin
Purpose
Array power for real-only output

Syntax
Z = realpow(X,Y)

Description
Z = 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 realpow is the set of all real numbers, i.e., all elements of the output array Z must be real.

Examples
X = -2*ones(3,3)

    X = 
     -2  -2  -2
     -2  -2  -2
     -2  -2  -2

Y = pascal(3)

    ans =
     1   1   1
     1   2   3
     1   3   6

    realpow(X,Y)

    ans =
     -2  -2  -2
     -2   4  -8
     -2  -8  64

See Also
reallog, realsqrt, .^ (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 =

<table>
<thead>
<tr>
<th>16</th>
<th>2</th>
<th>3</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

realsqrt(M)

ans =

<table>
<thead>
<tr>
<th>4.0000</th>
<th>1.4142</th>
<th>1.7321</th>
<th>3.6056</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2361</td>
<td>3.3166</td>
<td>3.1623</td>
<td>2.8284</td>
</tr>
<tr>
<td>3.0000</td>
<td>2.6458</td>
<td>2.4495</td>
<td>3.4641</td>
</tr>
<tr>
<td>2.0000</td>
<td>3.7417</td>
<td>3.8730</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

See Also
reallog, realpow, sqrt, sqrm
Purpose
Record data and event information to file

Syntax
record(obj)
record(obj,'switch')

Arguments
obj A serial port object.
'switch' Switch recording capabilities on or off.

Description
record(obj) toggles the recording state for 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, 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

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 (daspect). Note that the
horizontal and vertical curvature can be different. Also, note the effects of using a single value for 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 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])
```
ylim([1,13])

Object Hierarchy

Axes
  Group Object
    Rectangle
    Rectangle
Setting Default Properties

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

```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, rectangle properties`

“Object Creation Functions” on page 1-93 for related functions

See the `annotation` function for information about the rectangle annotation object.

`Rectangle Properties` for property descriptions
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.

**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**

functional 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)

```plaintext
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
    sel_typ = get(gcf,'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
functional handle, cell array containing function handle and additional arguments, or string (not recommended)
**Rectangle Properties**

*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 `g cbo`.

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

*Curvature*

one- or two-element vector \([x, y]\)
**Rectangle Properties**

*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**

functional 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.

```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.

**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.
Rectangle Properties

- 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 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 accidently 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 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
Rectangle Properties

The width of the rectangle edge line. Specify this value in points (1 point = \(\frac{1}{72}\) 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 \([x, y, width, height]\)

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 \(width\) and \(height\) 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 i-th row of A and the j-th row of B.

**Note** A position vector is a four-element vector \([x, y, \text{width}, \text{height}]\), 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 MATLAB file recycling option. 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 the PC or Macintosh, or a temporary directory on UNIX. (To locate this directory on UNIX, 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 recycling on for all of your MATLAB sessions using the Preferences dialog box (Select File > Preferences > General). Under the heading Default behavior of the delete function select Move files to the Recycle Bin.

S = recycle state sets the MATLAB recycle option to the given state, either on or off. Return value S shows the previous recycle state.

S = recycle('state') is the function format for this command.

Remarks
On UNIX systems, 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 system at 2:09:28 in the afternoon of November 9, 2004 would be copied to a directory named

/tmp/MATLAB_Files_09-Nov-2004_14_09_28

To set the recycle state for all MATLAB sessions, use the Preferences dialog box. Open the Preferences dialog and select General. To
enable or disable recycling, click Move files to the recycle bin or Delete files permanently. See “General Preferences for MATLAB” in the Desktop Tools and Development Environment documentation for more information.

You can recycle files that are stored on your local computer system, but not files that you access over a network connection. On Windows systems, when you use the delete function on files accessed over a network, MATLAB removes the file entirely.

**Examples**

Start from a state where file recycling has been turned off. Check the current recycle state:

```matlab
recycle
ans =
    off
```

Turn file recycling on. Delete a file and verify that it has been transferred to the recycle bin or temporary folder:

```matlab
recycle on;
delete myfile.txt
```

**See Also**
delete, dir, ls, fileparts, mkdir, rmdir
**Purpose**
Reduce number of patch faces

**Syntax**

```matlab
nfv = reducepatch(p,r)
fv = reducepatch(fv,r)
fv = reducepatch(p) or fv = reducepatch(fv)
reducepatch(...,'fast')
reducepatch(...,'verbose')
fv = reducepatch(fv,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. MATLAB 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 = reducepatch(f,v,r) performs the reduction on the faces in f and the vertices in v.

[ nf, nv ] = reducepatch(...) 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] = flow;
p = patch(isosurface(x, y, z, v, -3));
set(p,'facecolor','w','EdgeColor','b');
daspect([1,1,1])
view(3)
figure;
h = axes;
p2 = 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 1-101 for related functions

Vector Field Displayed with Cone Plots for another example
Purpose
Reduce number of elements in volume data set

Syntax

\[
\begin{align*}
[nx,ny,nz,nv] &= \text{reducevolume}(X,Y,Z,V,[Rx,Ry,Rz]) \\
[nx,ny,nz,nv] &= \text{reducevolume}(V,[Rx,Ry,Rz]) \\
vv &= \text{reducevolume}(\ldots)
\end{align*}
\]

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, MATLAB 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 \([m,n,p] = \text{size}(V)\).

\(nv = \text{reducevolume}(\ldots)\) 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 (\text{squeeze}) into three dimensions and then reduced (\text{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 (\text{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 (\text{patch}, \text{isosurface}, \text{isonormals}).

- A second patch (\(p2\)) with an interpolated face color draws the end caps (\text{FaceColor}, \text{isocaps}).

- The view of the object is set (\text{view}, \text{axis}, \text{daspect}).
reducevolume

- 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).

```matlab
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 1-101 for related functions
refresh

**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 1-94 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, MATLAB 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.

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.

Examples

This example creates a contour plot and changes its data source. The call to refreshdata causes the graph to update.

```matlab
z = peaks(5);
[c h] = contour(z,'ZDataSource','z');
drawnow
pause(3) % Wait 3 seconds and the graph will update
```
refreshdata

z = peaks(20);
refreshdata(h)

See Also

The [X,Y,Z]DataSource properties of plot objects.
**Purpose**

Match regular expression

**Syntax**

```matlab
regexp('str', 'expr')
[start end extents match tokens names] = regexp('str', 'expr')
[v1 v2 ...] = regexp('str', 'expr', q1, q2, ...)
[v1 v2 ...] = regexp('str', 'expr', ..., options)
```

Each of these syntaxes apply 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`:

```matlab
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.

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 2-2645.

[start end extents match tokens names] = 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.

```matlab
[v1 v2 ...] = regexp('str', 'expr', q1, q2, ...) 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>
</tbody>
</table>

\[v1 \ v2 \ldots\] = 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 2-2643 below.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</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 10.</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 6 illustrates the this mode.

<table>
<thead>
<tr>
<th>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 <code>regexp</code>).</td>
</tr>
<tr>
<td>'ignorecase'</td>
<td>(?i)</td>
<td>Do not consider letter case when matching patterns to a string. (The default for <code>regexpi</code>).</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 7 illustrates the Dot Matching mode.
### 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 8 illustrates the Anchor mode.

<table>
<thead>
<tr>
<th>Mode Keyword</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'stringanchors'</td>
<td>(?!m)</td>
<td>Match the ^ and $ metacharacters at the beginning and end of a string.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(This is the default).</td>
</tr>
<tr>
<td>'lineanchors'</td>
<td>(?!m)</td>
<td>Match the ^ and $ metacharacters at the beginning and end of a line.</td>
</tr>
</tbody>
</table>

### Spacing Mode

Use the Spacing mode to control how MATLAB interprets space characters and comments within the string being parsed. Example 9 illustrates the Spacing mode.
### 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 # 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 '\ ' and '#' to match space and # characters.)</td>
</tr>
</tbody>
</table>

### Remarks

See “Regular Expressions” in the MATLAB Programming 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';
```
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:

```matlab
s1{:}
ans =
   19
ans =
   11  1
ans =
   12  3  4  5  6
```

Space characters, ' \s ', were found at these `str` indices:

```matlab
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 — 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>'
str =
if <code>A</code> == x<sup>2</sup>, <em>disp(x)</em>
expr = '<(\w+).*?>.*?</\1>,'
[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 documentation for information on using tokens.

**Example 5 — 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{:}
an =
   'John'    'Davis'
an =
   'Rogers'  'James'
```

Now examine the `names` structure that was returned. First and last names appear in a more usable order:

```matlab
names(:,1)
an =
   first: 'John'
   last: 'Davis'
```
names(:,2)
ans =
  first: 'James'
  last: 'Rogers'

**Example 6 — 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')
anst =
  'case'
```

Now disable case-sensitive matching to find both instances of `case`:

```matlab
regexp(str, 'case', 'ignorecase', 'match')
anst =
  '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:

```matlab
M = regexp(str, {'(?-i)\w{5}(?=CASE)', ...
  '(?i)\w{5}(?=CASE)'}, 'match');
```

```matlab
M{:}
anst =
  'UPPER'
anst =
  'UPPER'  'lower'
```

**Example 7 — Using the Dot Matching Mode**

Parse the following string that contains a newline (\n) character:
str = sprintf('abc\ndef')
str =
    abc
def

When you use the default mode, dotall, MATLAB includes the newline in the characters matched:

regexp(str, '.', 'match')
ans =
    'a' 'b' 'c' [1x1 char] 'd' 'e' 'f'

When you use the dotexceptnewline mode, MATLAB skips the newline character:

regexp(str, '.', 'match', 'dotexceptnewline')
ans =
    'a' 'b' 'c' 'd' 'e' 'f'

**Example 8 — 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 stringanchors 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 lineanchors mode, MATLAB interprets $ as an end-of-line specifier, and finds the last two words of each line:
Example 9 — Using the Freesping Mode

Create a file called regexp_str.txt containing the following text. Because the first line enables freesping mode, MATLAB ignores all spaces and comments that appear in the file:

```matlab
(?x)   # turn on freesping.

# This pattern matches a string with a repeated letter.
\w*    # First, match any number of preceding word characters.

(       # Mark a token.
  \w    # Match a word character.
)      # Finish capturing said token.
\1     # Backreference to match what token #1 matched.
\w*    # Finally, match the remainder of the word.
```

Here is the string to parse:

```matlab
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:

```matlab
patt = fileread('regexp_str.txt');
regexp(str, patt, 'match')
ans =
      'Looking' 'letters' 'appear' 'succession'
```

Example 10 — Displaying Parsing Warnings

To help debug problems in parsing a string with regexp, regexpi, or regexprep, use the 'warnings' option to view all warning messages:
regexp, regexp

regexp('$.', '[a-]', 'warnings')
Warning: Unbound range.
[a-]
|

See Also
regexp, regexptranslate, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi

2-2652
**Purpose**
Replace string using regular expression

**Syntax**

\[
\text{s} = \text{regexprep('str', 'expr', 'repstr')} \\
\text{s} = \text{regexprep('str', 'expr', 'repstr' options)}
\]

**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 \( repstr \).

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 2-2654.

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 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>'ignorecase'</td>
<td>Ignore case when matching and when replacing.</td>
</tr>
</tbody>
</table>
### Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 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 `repstr` inputs follow these rules:

- If `expr` is a cell array of strings and `repstr` is a single string, `regexprep` uses the same replacement string on each expression in `expr`.
- If `expr` is a single string and `repstr` is a cell array of N strings, `regexprep` attempts to make N matches and replacements.
- If both `expr` and `repstr` are cell arrays of strings, then `expr` and `repstr` must contain the same number of elements, and `regexprep` pairs each `repstr` 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
>> regexprep(data, 'm[yA]', 'My
```
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:

regexprep(str, pat, 'April', 'preservecase')
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

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:

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 = regexprep(str, '(.)\1', '--', 'ignorecase')
a =
    'Whose w--ds 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.'

See Also
regexp, regexpi, regexptranslate, strfind, findstr, strmatch,
strcmp, strcmpi, strncmp, strncmpi
Purpose
Translate string into regular expression

Syntax
\[
s2 = \text{regexptranslate}(\text{type}, s1)\]

Description
\(s2 = \text{regexptranslate}(\text{type}, s1)\) 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 \text{regexp}. The \text{type} input can be either one of the following strings that define the type of translation to be performed.

<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 \text{regexp} and \text{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 \text{regexp} and \text{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 \text{regexp} interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\\' and 'n' in this string:

\[
\begin{align*}
\text{str} &= \text{'}\text{The sequence \n generates a new line'}\text{'}; \\
\text{pat} &= \text{'}\text{\n}'
\end{align*}
\]

\[
\begin{align*}
\text{regexp(str, pat)} \\
\text{ans} &= \text{[]}
\end{align*}
\]
To have regexp interpret the expression expr as the characters ‘\’ and ‘n’, first translate the expression using regexptranslate:

```
pat2 = regexptranslate('escape', pat)
pat2 =
  
regexp(str, pat2)
ans =
  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:

```
str = 'I walk up, they walked up, we are walking up.';
p = 'walk(\w*) up';

regexpprep(str, p, 'ascend$1')
ans =
  I ascend, they ascended, we are ascending.
```

Make another replacement on the same string, this time treating the ‘$1’ as literal characters:

```
regexpprep(str, p, regexptranslate('escape', 'ascend$1'))
ans =
  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:

```
files = ['test1.mat, myfile.mat, newfile.txt, ' ...
  'jan30.mat, table3.xls'];
regexp(str, regexptranslate('wildcard', '*.mat'), 'match')
ans =
```
To see the translation, you can type

```matlab
regexptranslate('wildcard','*.mat')
ans =
\w+.mat
```

**See Also**
regexp, regexpi, regexprep
registerevent

**Purpose**
Register event handler with control's event

**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 control 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).

**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
ans =
    Click = void Click()
    DblClick = void DblClick()
    MouseDown = void MouseDown(int16 Button, int16 Shift, Variant x, Variant y)
```

Register all events with the same event handler routine, `sampev`. Use `eventlisteners` to see the event handler used by each event:
h.registerevent('sampev');
h.eventlisteners
ans =
    {'click'   'sampev'}
    {'dblclick' 'sampev'}
    {'mousedown' 'sampev'}

h.unregisterallevents;

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.

h.registerevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});

h.eventlisteners
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:

```matlab
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200]);
registerevent(h, @sampev);
```

**Register Excel Workbook Events Example**

Create an Excel Workbook object.

```matlab
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
```

Register all events with the same event handler routine, AllEventHandler.
wb.registerevent('AllEventHandler')
wb.eventlisteners

MATLAB displays the list of all Workbook events, registered with AllEventHandler.

\[
\text{ans} = \\
\{'Open', 'AllEventHandler'\} \\
\{'Activate', 'AllEventHandler'\} \\
\{'Deactivate', 'AllEventHandler'\} \\
\{'BeforeClose', 'AllEventHandler'\} \\
\ldots \\
\ldots \\
\]

**See Also**

events, eventlisteners, unregisterevent, unregisterallevents, isevent
**Purpose**

Refresh function and file system path caches

**Syntax**

```
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 shadowed functions (functions that have been called but not cleared in the current session) 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. Therefore, use `rehash` with no arguments only when you run an M-file that updates another M-file, and the calling file needs to reuse the updated version before it 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. 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 using the most current versions of your M-files, run `rehash path`.

`rehash toolbox` updates all directories in `matlabroot/toolbox`. Run this when you add or remove files in `matlabroot/toolbox` during a session by some means other than MATLAB tools.

`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 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 Preferences > General.

See Also

addpath, clear, path, rmpath

“Toolbox Path Caching in MATLAB” in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**
Release 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.

**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 =
    Interface.Standard_OLE_Types.Font

TFont.Name = 'Viva BoldExtraExtended';
TFont.Bold = 0;
```

When you're finished working with the title font, release the `TitleFont` interface:
TFont.release;

Now create a GridFont interface and use it to modify the size of the calendar's date numerals:

```c
GFont = cal.GridFont
GFont =
    Interface.Standard_OLE_Types.Font
GFont.Size = 16;
```

When you're done, delete the cal object and the figure window:

```c
cal.delete;
delete(f);
clear f;
```

**See Also**
delete, save, load, actxcontrol, actxserver
**Purpose**
Remainder after division

**Syntax**

\[ R = \text{rem}(X,Y) \]

**Description**

\[ R = \text{rem}(X,Y) \] if \( Y \neq 0 \), returns \( X - n \cdot Y \) where \( n = \text{fix}(X./Y) \). 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:

- \( \text{rem}(X,0) \) is NaN
- \( \text{rem}(X,X) \) for \( X \neq 0 \) is 0
- \( \text{rem}(X,Y) \) for \( X \neq Y \) and \( Y \neq 0 \) has the same sign as \( X \).

**Remarks**

\( \text{mod}(X,Y) \) for \( X \neq Y \) and \( Y \neq 0 \) has the same sign as \( Y \).

\( \text{rem}(X,Y) \) and \( \text{mod}(X,Y) \) are equal if \( X \) and \( Y \) have the same sign, but differ by \( Y \) if \( X \) and \( Y \) have different signs.

The \( \text{rem} \) function returns a result that is between 0 and \( \text{sign}(X) \cdot \text{abs}(Y) \). If \( Y \) is zero, \( \text{rem} \) returns NaN.

**See Also**

\( \text{mod} \)
Purpose

Remove timeseries objects from tscollection object

Syntax

tsc = removets(tsc,Name)

Description

tsc = removets(tsc,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 tscollection.

1 Create two timeseries objects, ts1 and ts2.

   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');

2 Create a tscollection object tsc, which includes ts1 and ts2.

   tsc=tscollection({ts1 ts2});

3 To view the members of tsc, type the following at the MATLAB prompt:

   tsc

MATLAB responds with

   Time Series Collection Object: unnamed

   Time vector characteristics

   Start time      1 seconds
   End time        5 seconds

   Member Time Series Objects:
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=removets(tsc,'speed');

5 To view the current members of tsc, type the following at the MATLAB prompt:

    tsc

MATLAB responds with

    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
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.

```matlab
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 = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p...])

Description

B = 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 [size(A,1)*m, (size(A,2)*n]. The statement repmat(A,n) creates an n-by-n tiling.

B = repmat(A,[m n]) accomplishes the same result as repmat(A,m,n).

B = repmat(A,[m n p...]) produces a multidimensional array B composed of copies of A. The size of B is [size(A,1)*m, size(A,2)*n, size(A,3)*p, ...].

Remarks

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:

• repmat(NaN,m,n) returns the same result as NaN(m,n).
• repmat(single(inf),m,n) is the same as inf(m,n,'single').
• repmat(int8(0),m,n) is the same as zeros(m,n,'int8').
• repmat(uint32(1),m,n) is the same as ones(m,n,'uint32').
• repmat(eps,m,n) is the same as eps(ones(m,n)).

Examples

In this example, repmat replicates 12 copies of the second-order identity matrix, resulting in a “checkerboard” pattern.

B = repmat(eye(2),3,4)

B =  
1 0 1 0 1 0 1 0  
0 1 0 1 0 1 0 1  
1 0 1 0 1 0 1 0
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

\[
\begin{align*}
    ts &= \text{resample}(ts, \text{Time}) \\
    ts &= \text{resample}(ts, \text{Time}, \text{interp}_\text{method}) \\
    ts &= \text{resample}(ts, \text{Time}, \text{interp}_\text{method}, \text{code})
\end{align*}
\]

Description

\(ts = \text{resample}(ts,\text{Time})\) 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 \(ts.\text{TimeInfo}.\text{StartDate}\) property and in the same units that \(ts\) uses. The resample operation uses the default interpolation method, which you can view by using the \(\text{getinterpmethod}(ts)\) syntax.

\(ts = \text{resample}(ts,\text{Time},\text{interp}_\text{method})\) resamples the timeseries object \(ts\) using the interpolation method given by the string \(\text{interp}_\text{method}\). Valid interpolation methods include ‘linear’ and ‘zoh’ (zero-order hold).

\(ts = \text{resample}(ts,\text{Time},\text{interp}_\text{method},\text{code})\) resamples the timeseries object \(ts\) using the interpolation method given by the string \(\text{interp}_\text{method}\). 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.

\[
    ts = \text{timeseries}([1.1 \ 2.9 \ 3.7 \ 4.0 \ 3.0],1:5,'Name','speed');
\]

2. Transpose \(ts\) to make the data columnwise.

\[
    ts = \text{transpose}(ts)
\]

MATLAB displays

```
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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>2</td>
<td>2.9</td>
</tr>
<tr>
<td>3</td>
<td>3.7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</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])

MATLAB displays the resampled time series as follows:

Time Series Object: speed

Time vector characteristics

Length 5
Start time 1 seconds
End time 4.900000e+000 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>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
resample (tscollection)

**Purpose**
Select or interpolate data in `tscollection` using new time vector

**Syntax**
- `tsc = resample(tsc,Time)`
- `tsc = resample(tsc,Time,interp_method)`
- `tsc = resample(tsc,Time,interp_method,code)`

**Description**
- `tsc = resample(tsc,Time)` resamples the `tscollection` object `tsc` on the new `Time` vector. When `tsc` uses date strings and `Time` is numeric, `Time` is treated as numerical specified relative to the `tsc.TimeInfo.StartDate` property and in the same units that `tsc` uses. The `resample` method uses the default interpolation method for each time series member.
- `tsc = resample(tsc,Time,interp_method)` resamples the `tscollection` object `tsc` using the interpolation method given by the string `interp_method`. Valid interpolation methods include 'linear' and 'zoh' (zero-order hold).
- `tsc = resample(tsc,Time,interp_method,code)` resamples the `tscollection` object `tsc` 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 `tscollection` that consists of two `timeseries` members.

1. Create two `timeseries` objects.
   ```matlab
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');
```

2. Create a `tscollection` `tsc`.
   ```matlab
tsc=tscollection({ts1 ts2});
```
   The time vector of the collection `tsc` is [1:5], which is the same as for `ts1` and `ts2` (individually).
3 Get the interpolation method for acceleration by typing

    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

    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, tscollection
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

```matlab
get(0,'factory')
```

If h is a figure, MATLAB 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 1-99 for related functions
**Purpose**  
Reshape array

**Syntax**
- \( B = \text{reshape}(A,m,n) \)
- \( B = \text{reshape}(A,m,n,p,...) \)
- \( B = \text{reshape}(A,[m n p ...]) \)
- \( B = \text{reshape}(A,\ldots,[],\ldots) \)
- \( B = \text{reshape}(A,siz) \)

**Description**
- \( B = \text{reshape}(A,m,n) \) returns the \( m \)-by-\( n \) matrix \( B \) whose elements are taken column-wise from \( A \). An error results if \( A \) does not have \( m \times n \) elements.
- \( B = \text{reshape}(A,m,n,p,...) \) or \( B = \text{reshape}(A,[m n p ...]) \) returns an \( n \)-dimensional array with the same elements as \( A \) but reshaped to have the size \( m \)-by-\( n \)-by-\( p \)-by-\( \ldots \). The product of the specified dimensions, \( m \times n \times p \times \ldots \), must be the same as \( \text{prod(size}(A)) \).
- \( B = \text{reshape}(A,\ldots,[],\ldots) \) calculates the length of the dimension represented by the placeholder \([\]\), such that the product of the dimensions equals \( \text{prod(size}(A)) \). The value of \( \text{prod(size}(A)) \) must be evenly divisible by the product of the specified dimensions. You can use only one occurrence of \([\]\).
- \( B = \text{reshape}(A,siz) \) 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 \( \text{prod}(siz) \) must be the same as \( \text{prod(size}(A)) \).

**Examples**
Reshape a 3-by-4 matrix into a 2-by-6 matrix.

\[
A = \\
\begin{bmatrix}
1 & 4 & 7 & 10 \\
2 & 5 & 8 & 11 \\
3 & 6 & 9 & 12 \\
\end{bmatrix}
\]

\[
B = \text{reshape}(A,2,6)
\]

\[
B = \\
\begin{bmatrix}
1 & 4 & 7 & 10 & 13 & 16 \\
2 & 5 & 8 & 11 & 14 & 17 \\
3 & 6 & 9 & 12 & 15 & 18 \\
\end{bmatrix}
\]
B = reshape(A,2,[])

B =
1 3 5 7 9 11
2 4 6 8 10 12

See Also    shiftdim, squeeze
The colon operator :
Purpose
Convert between partial fraction expansion and polynomial coefficients

Syntax
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)

Description
The residue function converts a quotient of polynomials to pole-residue representation, and back again.

[r,p,k] = 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

\[
\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + \ldots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + \ldots + a_n}
\]

where \( b_j \) and \( a_j \) are the jth elements of the input vectors b and a.

[b,a] = 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, \( \text{resi2} \) 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

\[ [r, p, k] = \text{residue}(b,a) \]

\[ r = \]
\[-1.4167 \]
\[-0.6653 \]
\[1.3320 \]

\[ p = \]
\[1.5737 \]
\[-1.1644 \]
\[-0.4093 \]

\[ k = \]
\[-1.2500 \]

Now, convert the partial fraction expansion back to polynomial coefficients.

\[ [b,a] = \text{residue}(r,p,k) \]

\[ b = \]
\[-1.2500 \]
\[-0.7500 \]
\[0.5000 \]
\[-1.7500 \]

\[ a = \]
\[1.0000 \]
\[-0.0000 \]
\[-2.0000 \]
\[-0.7500 \]

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 MATLAB 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 MATLAB 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 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, pathdef, 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
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 1-97 for related functions
**Purpose**  
Plot colormap

![rgbplot](2-2691)

**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 1-97 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 separate three-dimensional ribbons using `X = 1:size(Y,1)`.

`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,width)` specifies the width of the ribbons. The default is 0.75.

`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.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
```
ribbon

```
z = peaks(x,y);  
ribbon(y,z)  
colormap hsv
```

**See Also**

plot, plot3, surface, waterfall

"Polygons and Surfaces" on page 1-89 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`.

**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 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:OSERROR
```

indicating the directory `myfiles` is not empty.

To remove `myfiles` and its contents, run

```
[stat, mess]=rmdir('myfiles','s')
```

and MATLAB returns
indicating myfiles and its contents were removed.

See Also

cd, copyfile, delete, dir, error, fileattrib, filebrowser, lasterror, mkdir, movefile
**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')
       ..
   rmdir(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, “Using Dynamic Field Names”
Purpose
Remove directories from MATLAB search path

GUI
Alternatives
As an alternative to the `rmpath` function, use the Set Path dialog box.
To open it, select **File > Set Path** in the MATLAB desktop.

Syntax
`rmpath('directory')`
`rmpath directory`

Description
`rmpath('directory')` removes the specified directory from the current MATLAB search path. Use the full pathname for `directory`.
`rmpath directory` is the command form of the syntax.

Examples
Remove `/usr/local/matlab/mytools` from the search path.

```
    rmpath /usr/local/matlab/mytools
```

See Also
`addpath`, `cd`, `dir`, `genpath`, `matlabroot`, `partialpath`, `path`, `pathdef`, `pathsep`, `pathtool`, `rehash`, `restoredefaultpath`, `savepath`, `what`

Search Path in the MATLAB Desktop Tools and Development Environment documentation
Purpose

Remove preference

Syntax

rmpref('group','pref')
rmpref('group',{pref1,pref2,...,prefn})
rmpref('group')

Description

rmpref('group','pref') removes the preference specified by group and pref. It is an error to remove a preference that does not exist.

rmpref('group',{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.

rmpref('group') removes all the preferences for the specified group. It is an error to remove a group that does not exist.

Examples

addpref('mytoolbox','version','1.0')
rmpref('mytoolbox')

See Also

addpref, getpref, ispref, setpref, uigetpref, uisetpref
Purpose
Root object properties

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.

**CaptureMatrix**

(obsolete)

This property has been superseded by the `getframe` command.
CaptureRect
  (obsolete)

  This property has been superseded by the getframe 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

  `figure(h)`

  which restacks the figure to the top of the screen, or

  `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 \texttt{gcf} always returns a figure handle, and creates one if there are no figure objects.

\textbf{DeleteFcn}

\texttt{string}

This property is not used, because you cannot delete the root object.

\textbf{Diary}

\texttt{on | {off}}

\textit{Diary file mode}. When this property is on, MATLAB maintains a file (whose name is specified by the \texttt{DiaryFile} property) that saves a copy of all keyboard input and most of the resulting output. See also the \texttt{diary} command.

\textbf{DiaryFile}

\texttt{string}

\textit{Diary filename}. The name of the diary file. The default name is \texttt{diary}.

\textbf{Echo}

\texttt{on | {off}}

\textit{Script echoing mode}. When \texttt{Echo} is on, MATLAB displays each line of a script file as it executes. See also the \texttt{echo} command.

\textbf{ErrorMessage}

\texttt{string}

\textit{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

**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

\[
\begin{align*}
x &= \text{primary monitor width} + 1 \\
y &= \text{primary monitor height} + 1 
\end{align*}
\]

Querying the value of the figure MonitorPosition on a multiheaded system returns the position for each monitor on a separate line.

\[
v = \text{get}(0,'MonitorPosition')
v = \\]
\[
\begin{align*}
x & \ y & \text{width} & \text{height} & \% \text{ Primary monitor} \\
x & \ y & \text{width} & \text{height} & \% \text{ Secondary monitor} 
\end{align*}
\]

Note that MATLAB sets the value of the ScreenSize property to the combined size of the monitors.

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.
Root Properties

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,

[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 ScreenPixelPerInch 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 Windows task bar.

**Selected**

```markdown
on  |  off
```

This property has no effect on the root level.

**SelectionHighlight**

```markdown
{on}  |  off
```

This property has no effect on the root level.

**ShowHiddenHandles**

```markdown
on  |  {off}
```

*Show or hide handles marked as hidden.* When set to on, this property enables 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**

```markdown
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**

```markdown
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 $p = \text{poly}(r)$, 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 as
$$p = [1 \ -6 \ -72 \ -27]$$
The roots of this polynomial are returned in a column vector by
$$r = \text{roots}(p)$$
$$r =$$
$$12.1229$$
$$\quad -5.7345$$
$$\quad -0.3884$$

Algorithm
The algorithm simply involves computing the eigenvalues of the companion matrix:
$$A = \text{diag(ones(n-1,1),-1)};$$
$$A(1,:) = -c(2:n+1)/c(1);$$
$$\text{eig}(A)$$
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

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

rose
rose(theta)
rose(theta,x)
rose(theta,nbins)
rose(axes_handle,...)
h = rose(...)
[tout,rout] = rose(...)

Description

rose creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range. Each group is shown as one bin.

rose(theta) plots an angle histogram 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).

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.

    theta = 2*pi*rand(1,50);
    rose(theta)
See Also

compass, feather, hist, line, polar

“Histograms” on page 1-89 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 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
  -192    -43     49     44   -599    411    208    208
  -192     -8     8    59    208    208    99    -911
   29    -44    52   -23    208    208   -911     99
**Purpose**
Rotate matrix 90 degrees

**Syntax**
- \( B = \text{rot90}(A) \)
- \( B = \text{rot90}(A,k) \)

**Description**
- \( B = \text{rot90}(A) \) rotates matrix \( A \) counterclockwise by 90 degrees.
- \( B = \text{rot90}(A,k) \) rotates matrix \( A \) counterclockwise by \( k \times 90 \) degrees, where \( k \) is an integer.

**Examples**
The matrix

\[
X = \\
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\]

rotated by 90 degrees is

\[
Y = \text{rot90}(X) \\
Y = \\
3 & 6 & 9 \\
2 & 5 & 8 \\
1 & 4 & 7 \\
\]

**See Also**
flipdim, fliplr, flipud
**Purpose**  
Rotate object in specified direction

**Syntax**  
\[ \text{rotate}(h, \text{direction}, \alpha) \]
\[ \text{rotate}(\ldots, \text{origin}) \]

**Description**  
The `rotate` function rotates a graphics object in three-dimensional space, according to the right-hand rule.

\[ \text{rotate}(h, \text{direction}, \alpha) \] rotates the graphics object \( h \) by \( \alpha \) degrees. \text{direction} is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

\[ \text{rotate}(\ldots, \text{origin}) \] 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.

![Diagram of spherical coordinates](image)

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 1-99 for related functions
Rotating a 3-D View Using the Mouse

**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` 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.

**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. This read-only property 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.

**ButtonDownFilter <function_handle>**

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 object (empty in this release).
    % RES       a logical flag to determine whether the rotate operation should take place or the 'ButtonDownFcn' property of the object should take precedence.
```

**ActionPreCallback <function_handle>**

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 callbacks):

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

The event object has the following read-only property:

**Axes** The handle of the axes that is being rotated.

**ActionPostCallback <function_handle>**
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 callbacks):

```matlab
function myfunction(obj,event_obj)
% obj handle to the figure that has been clicked on.
% event_obj handle to event object. The object has the same properties as the EVENT_OBJ of the 'ActionPreCallback' callback.

flags = isAllowAxesRotate(h,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.

```matlab
setAllowAxesRotate(h,axes,flag)
```

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:

```matlab
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.

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.

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
**Purpose**
Round to nearest integer

**Syntax**
\[ Y = \text{round}(X) \]

**Description**
\[ Y = \text{round}(X) \] rounds the elements of \( X \) to the nearest integers. For complex \( X \), the imaginary and real parts are rounded independently.

**Examples**
\[ a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i] \]

\[
\begin{align*}
a &= \\
Columns 1 through 4 \\
-1.9000 & -0.2000 & 3.4000 & 5.6000 \\
Columns 5 through 6 \\
7.0000 & 2.4000 + 3.6000i
\end{align*}
\]

\[ \text{round}(a) \]

\[
\begin{align*}
\text{ans} &= \\
Columns 1 through 4 \\
-2.0000 & 0 & 3.0000 & 6.0000 \\
Columns 5 through 6 \\
7.0000 & 2.0000 + 4.0000i
\end{align*}
\]

**See Also**
ceil, fix, floor
**Purpose**

Reduced row echelon form

**Syntax**

\[
R = \text{rref}(A) \\
[R,jb] = \text{rref}(A) \\
[R,jb] = \text{rref}(A,\text{tol})
\]

**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

\[
\begin{align*}
0 & \quad 0 & \quad 1 & \quad -3 \\
0 & \quad 0 & \quad 0 & \quad 0
\end{align*}
\]

**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 \( U^*U = \text{eye(size}(A)) \). 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.

\[
u = \]

rsf2csf

-0.4916  -0.2756 - 0.4411i  0.2133 + 0.5699i  -0.3428
-0.4980  -0.1012 + 0.2163i  -0.1046 + 0.2093i  0.8001
-0.6751   0.1842 + 0.3860i  -0.1867 - 0.3808i  -0.4260
-0.2337  0.2635 - 0.6481i  0.3134 - 0.5448i  0.2466

T =

4.8121  -0.9697 + 1.0778i  -0.5212 + 2.0051i  -1.0067
  0   1.9202 + 1.4742i    2.3355   0.1117 + 1.6547i
  0    0          1.9202 - 1.4742i  0.8002 + 0.2310i
  0    0        0                     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.
Values for content | Description
--- | ---
varlist | Save only those variables that are in varlist. You can use the * wildcard to save only those variables that match the specified pattern. For example, save('A*') saves all variables that start with A.

-\texttt{regexp exprlist} | Save those variables that match any of the regular expressions in exprlist.

-\texttt{struct s} | Save as individual variables all fields of the scalar structure s.

-\texttt{struct s fieldlist} | Save as individual variables only the specified fields of structure s.

In this table, the terms \texttt{varlist}, \texttt{exprlist}, and \texttt{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 firstname lastname street town
```

Function format:

```
save('firstname', 'lastname', 'street', 'town')
```

\texttt{save filename \textit{options}} stores all variables from the MATLAB workspace in file \texttt{filename} according to one or more of the following options. If \texttt{filename} is not specified, MATLAB stores the data in a file called \texttt{matlab.mat}.  

2-2737
**Values for options** | **Description**
---|---
-**append** | Add new variables to those already stored in an existing MAT-file.
-**format** | Save using the specified binary or ASCII format. See the section on, “MAT-File Format Options” on page 2-2738, 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 2-2739, 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><strong>MAT-file format Options</strong></th>
<th><strong>How Data Is Stored</strong></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. 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.)

| version Option | Use When Running ... | To Save a MAT-File That You Can Load In ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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>MATLAB Versions</th>
<th>Data Items or Features Supported</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>MATLAB Versions</td>
<td>Data Items or Features Supported</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------</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 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:

```plaintext
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 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 variables vol and temp in ASCII format to a file named june10:

    save('d:\mymfiles\june10','vol','temp','-ASCII')

Example 4
Save the fields of structure s1 as individual variables rather than as an entire structure.

    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:

    whos -file newstruct.mat

    Name       Size     Bytes   Class          Attributes
    a           1x1      8        double array
    b           1x2      158      cell array
    c           1x6      12        char array

    Grand total is 16 elements using 178 bytes

Read only the b field into the MATLAB workspace.

    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:

    save('mydata', '-regexp', '^Mon|^Tue|^Wed');

Here is another way of doing the same thing. In this case, there are three separate expression arguments:
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 c1 x y
save c2 x y -compress
```

d1 = dir('c1.mat');
d2 = dir('c2.mat');

d1.bytes
ans =
   45000240 % Size of the uncompressed data
d2.bytes
ans =
   11985634 % Size of the compressed data

d2.bytes/d1.bytes
ans =
   0.2663 % Ratio of compressed to uncompressed

See Also
load, clear, diary, fprintf, fwrite, genvarname, who, whos, workspace, regexp
save (COM)

**Purpose**
Serialize control object to file

**Syntax**

- `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.

**Examples**

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
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:

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```
h.load('mwsample');
h.get
ans =
    Label: 'Label'
  Radius: 20
```

**See Also**
load, actxcontrol, actxserver, release, delete

2-2744
Purpose
Save serial port objects and variables to MAT-file

Syntax
save filename
save filename obj1 obj2...

Arguments
filename
The MAT-file name.
obj1
Serial port objects or arrays of serial port objects.
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

    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`.

```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. Note that 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

<table>
<thead>
<tr>
<th>ext</th>
<th>Value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcx</td>
<td></td>
<td>Paintbrush 24-bit</td>
</tr>
<tr>
<td>pgm</td>
<td></td>
<td>Portable Graymap</td>
</tr>
<tr>
<td>png</td>
<td></td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>ppm</td>
<td></td>
<td>PortablePixmap</td>
</tr>
<tr>
<td>tif</td>
<td></td>
<td>TIFF image, compressed</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 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.

If you want to control the size or resolution of figures saved in image (bitmapped) 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.

```
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.

```
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.

```
saveas(gcf,'logo', 'ill')
```

**Example 3: Specify File Format and Extension**

Save the current figure to the file star.eps using the Level 2 Color PostScript format. If you use doc print or help print, you can see from the table for print device types that the device type for this format is -dpsc2. The file created is star.eps.

```
saveas(gcf,'star.eps', 'psc2')
```

In another example, save the current Simulink block diagram to the file 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 -dtiffn. The file created is trans.tiff.

```
saveas(gcf,'trans.tiff', 'tiffn')
```

**See Also**

hgsave, open, print
“Printing” on page 1-91 for related functions

Simulink users, see also save_system
Purpose
User-defined extension of save function for user objects

Syntax
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 saveobj method for the object’s class, if such a method exists. The return value B is subsequently used by save to populate the MAT-file.

When you issue a save command on an object, MATLAB looks for a method called saveobj in the class directory. You can overload this method to modify the object before the save operation. For example, you could define a saveobj method that saves related data along with the object.

Remarks
saveobj can be overloaded only for user objects. save will not call saveobj for a built-in datatype, such as double, even if @double/saveobj exists.

saveobj will be separately invoked for each object to be saved.

A child object does not inherit the saveobj method of its parent class. To implement saveobj for any class, including a class that inherits from a parent, you must define a saveobj method within that class directory.

Examples
The following example shows a saveobj method written for the portfolio class. The method determines if a portfolio object has already been assigned an account number from a previous save operation. If not, saveobj calls getAccountNumber to obtain the number and assigns it to the account_number field. The contents of b is saved to the MAT-file.

```matlab
function b = saveobj(a)
if isempty(a.account_number)
    a.account_number = getAccountNumber(a);
end
b = a;
```
saveobj

See Also  save, load, loadobj
Purpose
Save current MATLAB search path to pathdef.m file

GUI Alternatives
As an alternative to the savepath function, use the Set Path dialog box. To open it, select File > Set Path in the MATLAB desktop.

Syntax
savepath
savepath newfile

Description
savepath saves the current MATLAB search path to pathdef.m. It returns

| 0 | If the file was saved successfully |
| 1 | If the save failed |

savepath newfile saves the current MATLAB search path to newfile, where newfile is in the current directory or is a relative or absolute path.

Examples
The statement

    savepath myfiles/pathdef.m

saves the current search path to the file pathdef.m, which is located in the myfiles directory in the MATLAB current directory.

Consider using savepath in your MATLAB finish.m file to save the path when you exit MATLAB.

See Also
addpath, cd, dir, finish, genpath, matlabroot, partialpath, pathdef, pathsep, pathtool, rehash, restoredefaultpath, rmpath, savepath, startup, what

Search Path 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**

```matlab
scatter(X,Y,S,C)
scatter(X,Y)
scatter(X,Y,S)
scatter(...,markertype)
scatter(...,'filled')
scatter(...,'PropertyName',propertyvalue)
scatter(axes_handles,...)
```

```matlab
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 `length(X)`-by-3 matrix, it specifies the colors of the markers as RGB values. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers).

`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).

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 1-90 for related functions

See Triangulation and Interpolation of Scatter Data for related information.

See Scattergroup Properties for property descriptions.
Purpose

3-D 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

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 colors of each marker. When C is a vector the same length as X, Y, and Z, the values in C are linearly mapped to the colors in the current colormap. When C is a length(X)-by-3 matrix, it specifies the colors of the markers as RGB values. 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.
scatter3(X,Y,Z,S) draws markers at the specified sizes (S) in a single color.
scatter3(...,markertype) uses the marker type specified instead of 'o' (see LineSpec for a list of marker specifiers).

scatter3(...,'filled') fills the markers.

scatter3(...,'PropertyName',propertyvalue) creates the scatter graph, applying the specified property settings. See scattergroup properties for a description of properties.

h = scatter3(...) returns handles to the scattergroup objects created by scatter3. See Scattergroup Properties for property descriptions.

**Backward-Compatible Version**

hpatch = scatter3('v6',...) returns the handles to the patch objects created by scatter3 (see Patch for a list of properties you can specify using the object handles and set).

**Remarks**

Use plot3 for single color, single marker size 3-D scatter plots.

**Examples**

```matlab
[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)
```
See Also

scatter, plot3

See Scattergroup Properties for property descriptions

“Scatter/Bubble Plots” on page 1-90 for related functions
## Scattergroup Properties

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Define scattergroup properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modifying Properties</strong></td>
<td>You can set and query graphics object properties using the set and get commands or the Property Editor (<code>propertyeditor</code>). Note that you cannot define default property values for scattergroup objects. See Plot Objects for information on scattergroup objects.</td>
</tr>
<tr>
<td><strong>Scattergroup Property Descriptions</strong></td>
<td>This section provides a description of properties. Curly braces `{} enclose default values.</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 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 interrupted 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.
Scattergroup Properties

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:

```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.

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.
Scattergroup Properties

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

*Label used by plot legends.* The legend function, the figure’s active legend, and the plot browser use this text when displaying labels for this object.

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.

**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
Scattergroup Properties

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.

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 theLineStyle 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>
</tbody>
</table>
### Scattergroup 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).

**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

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.

\[ t = \text{area}(Y, \text{'Tag', 'area1'}) \]
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)

*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 scatter function calling syntax assigns values to 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.
Scattergroup Properties

YData
scalar, vector, or matrix

Y-coordinates of scatter markers. The scatter function draws individual markers at each y-axis location in the YData array.

The input argument y in the scatter 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.

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)

*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)
```
ans =
1.0000  -7.1119  -815.8706
0       2.0000  -55.0236
0       0       3.0000

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, ZUNGTR, ZSTEQR (with output U)</td>
</tr>
<tr>
<td>Non-Hermitian</td>
<td>ZGEHRD, ZHSEQR</td>
</tr>
<tr>
<td></td>
<td>ZGEHRD, ZUNGHR, 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:
**Matrix A** | **Routine**  
---|---  
Real symmetric | SSYTRD, SSTEQR  
| SSYTRD, SORGTR, SSTEQR (with output U)  
Real nonsymmetric | SGEHRD, SHSEQR  
| SGEHRD, SORGHR, SHSEQR (with output U)  
Complex Hermitian | CHETRD, CSTEQR  
| CHETRD, CUNGTR, CSTEQR (with output U)  
Non-Hermitian | CGEHRD, CHSEQR  
| CGEHRD, CUNGR, CHSEQR (with output U)  

**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 \( \sec(\pi/2) \) does not evaluate as infinite but as the reciprocal of the floating-point accuracy \( \epsilon_\text{p} \), because \( \pi \) is a floating-point approximation to the exact value of \( \pi \).

**Definition**

The secant can be defined as

\[
\sec(z) = \frac{1}{\cos(z)}
\]

**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](http://www.netlib.org).

**See Also**

\( \text{secd}, \text{sech}, \text{asec}, \text{asecd}, \text{asech} \)
**Purpose**  
Secant of argument in degrees

**Syntax**  
Y = secd(X)

**Description**  
Y = secd(X) is the secant of the elements of X, expressed in degrees. For odd integers n, secd(n*90) is infinite, whereas 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 $-2\pi \leq x \leq 2\pi$.

    x = -2*pi:0.01:2*pi;
    plot(x,sech(x)), grid on
sech

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 1-99 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)
semilogy(...)
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
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.

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;
semilogy(x,10.^x)
semilogx, semilogy

See Also

line, LineSpec, loglog, plot

“Basic Plots and Graphs” on page 1-85 for related functions
Purpose

Return list of events control can trigger

Note  Support for send will be removed in a future release of MATLAB. Use the events function instead of send.
**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 2-2791. 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. MATLAB 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

```plaintext
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.

```plaintext
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
serial

Purpose
Create serial port object

Syntax
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)

Arguments
'port' The serial port name.
'PropertyName' A serial port property name.
PropertyValue A property value supported by PropertyName.
obj The serial port object.

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.

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.

```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.

```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`

#### Properties

`Name, Port, Status, Type`
**serialbreak**

**Purpose**  
Send break to device connected to serial port

**Syntax**  
serialbreak(obj)  
serialbreak(obj, time)

**Arguments**  
obj  
A serial port object.  

time  
The duration of the break, in milliseconds.

**Description**  
serialbreak(obj) sends a break of 10 milliseconds to the device connected to obj.

serialbreak(obj, time) sends a break to the device with a duration, in milliseconds, specified by time. Note that the duration of the break might be inaccurate under some operating systems.

**Remarks**  
For some devices, the break signal provides a way to clear the hardware buffer.

Before you can send a break to 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 to send a break while obj is not connected to the device.

serialbreak is a synchronous function, and blocks the command line until execution is complete.

If you issue serialbreak while data is being asynchronously written, an error is returned. In this case, you must call the stopasync function or wait for the write operation to complete.

**See Also**  
**Functions**  
open, stopasync

**Properties**  
Status
**Purpose**

Set object properties

**Syntax**

- `set(H,'PropertyName',PropertyValue,...)`
- `set(H,a)`
- `set(H,pn,pv,...)`
- `set(H,pn,<m-by-n cell array>)`
- `a = set(h)`
- `a = set(h,'Default')`
- `a = set(h,'DefaultObjectTypePropertyName')`
- `pv = set(h,'PropertyName')`

**Description**

`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,<m-by-n cell array>)` 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, MATLAB displays the information on the screen. `h` must be scalar.
a = set(h, 'Default') returns the names of properties having default values set on the object identified by h. set also returns the possible values if they are strings. h must be scalar.

a = set(h, 'DefaultObjectTypePropertyName') returns the possible values of the named property for the specified object type, if the values are strings. The argument DefaultObjectTypePropertyName is the word Default concatenated with the object type (e.g., axes) and the property name (e.g., CameraPosition). For example, DefaultAxesCameraPosition. h must be scalar.

pv = set(h, 'PropertyName') returns the possible values for the named property. If the possible values are strings, set returns each in a cell of the cell array pv. For other properties, set returns an empty cell array. 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 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 MATLAB 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,

```matlab
f = figure('Units','characters',...    'Position',[30 30 120 35]);
```

Examples

Set the Color property of the current axes to blue.

```matlab
set(gca,'Color','b')
```

Change all the lines in a plot to black.

```matlab
plot(peaks)
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 `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 colors and enables the controls.

```matlab
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,

```matlab
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).

```matlab
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`,

```matlab
set(H,PropName,PropVal)
```
where \( \text{length}(H) = 3 \) 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

“Finding and Identifying Graphics Objects” on page 1-92 for related functions
**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” in the External Interfaces documentation for information on how MATLAB converts workspace matrices to COM data types.

**Examples**
Create an `mwsamp` control and use `set` to change the `Label` and `Radius` properties:

```matlab
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`:

```matlab
h.Label = 'Click to fire event';
h.Radius = 40;
h.Redraw;
```

**See Also**
- `get`, `inspect`, `isprop`, `addproperty`, `deleteproperty`
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)

Arguments

obj A serial port object or an array of serial port objects.
'PropertyName' A property name for obj.
PropertyValue A property value supported by PropertyName.
PN A cell array of property names.
PV A cell array of property values.
S A structure with property names and property values.
props A structure array whose field names are the property names for obj, or cell array of possible values.

Description

set(obj) displays all configurable properties values for 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`.

```matlab
s = serial('COM1');
set(s,'BaudRate',9600,'Parity','even')
set(s,{'StopBits','RecordName'},[2,'sydney.txt'])
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

See Also

Functions

get
Purpose
Configure or display timer object properties

Syntax

```
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(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.

```matlab
t = timer;
```

Display all configurable properties and their possible values.

```matlab
set(t)
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyMode</td>
<td>{drop}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>string -or- function handle -or- cell array</td>
</tr>
<tr>
<td>ExecutionMode</td>
<td>{singleShot}</td>
</tr>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>ObjectVisibility</td>
<td>{on}</td>
</tr>
<tr>
<td>Period</td>
<td></td>
</tr>
<tr>
<td>StartDelay</td>
<td></td>
</tr>
<tr>
<td>StartFcn</td>
<td>string -or- function handle -or- cell array</td>
</tr>
<tr>
<td>StopFcn</td>
<td>string -or- function handle -or- cell array</td>
</tr>
<tr>
<td>Tag</td>
<td></td>
</tr>
<tr>
<td>TasksToExecute</td>
<td></td>
</tr>
<tr>
<td>TimerFcn</td>
<td>string -or- function handle -or- cell array</td>
</tr>
<tr>
<td>UserData</td>
<td></td>
</tr>
</tbody>
</table>

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)`
setabstime (timeseries)

**Purpose**
Set times of timeseries object as date strings

**Syntax**

```
-ts = setabstime(ts,Times)
ts = setabstime(ts,Times,Format)
```

**Description**

`ts = setabstime(ts,Times)` 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.

`ts = setabstime(ts,Times,Format)` explicitly specifies the date-string format used in `Times`.

**Examples**

1. Create a time-series object.
   ```
ts = timeseries(rand(3,1))
   ```

2. Set the absolute time vector.
   ```
   ```

**See Also**
`datestr`, `getabstime (timeseries)`, `timeseries`
Purpose

Set times of tscollection object as date strings

Syntax

\[
tsc = \text{setabstime}(tsc, \text{Times}) \\
tsc = \text{setabstime}(tsc, \text{Times}, \text{format})
\]

Description

\[
tsc = \text{setabstime}(tsc, \text{Times}) \text{ sets the times in tsc using the date} \\
\text{strings Times. Times must be either a cell array of strings, or a char} \\
\text{array containing valid date or time values in the same date format.}
\]

\[
tsc = \text{setabstime}(tsc, \text{Times}, \text{format}) \text{ specifies the date-string} \\
\text{format used in Times explicitly.}
\]

Examples

1 Create a tscollection object.

\[
tsc = \text{tscollection(timeseries(rand(3,1)))}
\]

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.

**See Also**  
`getappdata`, `isappdata`, `rmappdata`
Purpose
Find set difference of two vectors

Syntax
c = setdiff(A, B)
c = setdiff(A, B, 'rows')
[c,i] = setdiff(...)

Description
c = setdiff(A, B) returns the values in A that are not in B. In set theory terms, c = A - 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 = setdiff(A, B, 'rows'), when A and B are matrices with the same number of columns, returns the rows from A that are not in B.

[c,i] = setdiff(...) also returns an index vector index such that c = a(i) 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 = magic(5);
B = magic(4);
[c, i] = setdiff(A(:,), B(:,));
c' =  17  18  19  20  21  22  23  24  25
i' =   1  10  14  18  19  23  2  6  15

See Also
intersect, ismember, issorted, setxor, union, unique
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. Under the Windows operating system, this is equivalent to undefining the variable. On most UNIX-like 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:

    setenv('TEMP', 'C:\TEMP');
    getenv('TEMP')

% Append the Perl\bin directory to your system PATH variable:

    setenv('PATH', [getenv('PATH') ';D:\Perl\bin']);

See Also
genv, system, unix, dos, !
Purpose

Set value of structure array field

Syntax

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, MATLAB 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 “Naming conventions for Structure Field Names” for guidelines to 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 \texttt{setfield} with variable and quoted field names and additional subscripting arguments.

\begin{verbatim}
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);
\end{verbatim}

You can check the outcome using the standard structure syntax.

\begin{verbatim}
grades(class).John_Doe.Math(10, 21:30)
\end{verbatim}

ans =

\begin{verbatim}
  85  89  76  93  85  91  68  84  95  73
\end{verbatim}

\textbf{See Also} \texttt{getfield}, \texttt{fieldnames}, \texttt{isfield}, \texttt{orderfields}, \texttt{rmfield}, “Using Dynamic Field Names”
Purpose
Set default interpolation method for timeseries object

Syntax
\[ \text{ts} = \text{setinterpmethod}(\text{ts}, \text{Method}) \]
\[ \text{ts} = \text{setinterpmethod}(\text{ts}, \text{FHandle}) \]
\[ \text{ts} = \text{setinterpmethod}(\text{ts}, \text{InterpObj}) , \]

Description
\[ \text{ts} = \text{setinterpmethod}(\text{ts}, \text{Method}) \] sets the default interpolation method for timeseries object \( \text{ts} \), where \( \text{Method} \) is a string. \( \text{Method} \) in \( \text{ts} \). \( \text{Method} \) is either ‘linear’ or ‘zoh’ (zero-order hold). For example:

\[
\begin{align*}
\text{ts} &= \text{timeseries}(\text{rand}(100,1),1:100) \\
\text{ts} &= \text{setinterpmethod}(\text{ts},'\text{zoh}')
\end{align*}
\]

\[ \text{ts} = \text{setinterpmethod}(\text{ts}, \text{FHandle}) \] sets the default interpolation method for timeseries object \( \text{ts} \), where \( \text{FHandle} \) is a function handle to the interpolation method defined by the function handle \( \text{FHandle} \). For example:

\[
\begin{align*}
\text{ts} &= \text{timeseries}(\text{rand}(100,1),1:100) \\
\text{myFuncHandle} &= @(\text{new\_Time},\text{Time},\text{Data})... \\
&\quad \quad \text{interp1}((\text{Time},\text{Data},\text{new\_Time},...
&\quad \quad \text{'linear'},'\text{extrap}') \\
\text{ts} &= \text{setinterpmethod}(\text{ts},\text{myFuncHandle}) \\
\text{ts} &= \text{resample}(\text{ts},[-5:0.1:10]) \\
\text{plot}(\text{ts})
\end{align*}
\]

Note
For \( \text{FHandle} \), you must use three input arguments. The order of input arguments must be \( \text{new\_Time}, \text{Time}, \text{and Data} \). The single output argument must be the interpolated data only.

\[ \text{ts} = \text{setinterpmethod}(\text{ts}, \text{InterpObj}) , \] where \( \text{InterpObj} \) is a \( \text{tsdata.interpolation} \) object that directly replaces the interpolation object stored in \( \text{ts} \). For example:

\[
\begin{align*}
\text{ts} &= \text{timeseries}(\text{rand}(100,1),1:100)
\end{align*}
\]
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
Purpose
Set component position in pixels

Syntax
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]);
```
See Also  getpixelposition, uicontrol, uipanel
Purpose

Set preference

Syntax

```matlab
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.

```matlab
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

```matlab
addpref('mytoolbox','version','0.0')
setpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')
```

```matlab
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>
settimeseriesnames

**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
\[ c = \text{setxor}(A, B) \]
\[ c = \text{setxor}(A, B, 'rows') \]
\[ [c, ia, ib] = \text{setxor}(...) \]

Description
\( c = \text{setxor}(A, B) \) 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.

\( c = \text{setxor}(A, B, 'rows') \), when \( A \) and \( B \) are matrices with the same number of columns, returns the rows that are not in the intersection of \( A \) and \( B \).

\([c, ia, ib] = \text{setxor}(...)\) 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
\[ a = [-1 0 1 Inf -Inf NaN]; \]
\[ b = [-2 pi 0 Inf]; \]
\[ c = \text{setxor}(a, b) \]
\[ c = \]
\[-Inf -2.0000 -1.0000 1.0000 3.1416 NaN \]

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.

```
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, mesh, patch, pcolor, surf

The EdgeColor and FaceColor properties for patch and surface graphics objects.

“Color Operations” on page 1-97 for related functions
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
Purpose
Show or hide figure plot tool

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 the size of patch faces

Syntax
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, MATLAB 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 1-101 for related functions
Purpose  Signum function

Syntax  \( Y = \text{sign}(X) \)

Description  \( Y = \text{sign}(X) \) 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 \( X \), \( \text{sign}(X) = X./\text{abs}(X) \).

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 \( -\pi \leq x \leq \pi \).

\[
x = -\pi:0.01:\pi; \\
plot(x,\sin(x)), \text{ grid on}
\]

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
\(\text{sin} \) 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.

See Also
sind, sinh, asin, asind, asinh
Purpose  
Sine of argument in degrees

Syntax  
Y = sind(X)

Description  
Y = sind(X) is the sine of the elements of X, expressed in degrees. For integers n, sind(n*180) is exactly zero, whereas sin(n*pi) reflects the accuracy of the floating point value of pi.

See Also  
sin, sinh, asin, asind, asinh
Purpose  Convert to single precision

Syntax  \[B = \text{single}(A)\]

Description  \(B = \text{single}(A)\) 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, \text{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

\[
\begin{align*}
a &= \text{magic}(4); \\
b &= \text{single}(a);
\end{align*}
\]

\[
\text{whos}
\]

\begin{tabular}{lllll}
Name & Size & Bytes & Class \\
\hline
   a & 4x4 & 128 & double array \\
b & 4x4 & 64 & single array \\
\end{tabular}

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**
- \( d = \text{size}(X) \)
- \([m,n] = \text{size}(X)\)
- \(m = \text{size}(X,\text{dim})\)
- \([d1,d2,d3,\ldots,dn] = \text{size}(X),\)

**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 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 \(\text{dim}\).
- \([d1,d2,d3,\ldots,dn] = \text{size}(X),\) for \(n > 1\), returns the sizes of the dimensions of the array \(X\) in the variables \(d1,d2,d3,\ldots,dn,\) provided the number of output arguments \(n\) equals \(\text{ndims}(X)\). If \(n\) does not equal \(\text{ndims}(X)\), the following exceptions hold:
  - \(n < \text{ndims}(X)\) \(d_i\) equals the size of the \(i\)th dimension of \(X\) for \(1 \leq i < n\), but \(dn\) equals the product of the sizes of the remaining dimensions of \(X\), that is, dimensions \(n\) through \(\text{ndims}(X)\).
  - \(n > \text{ndims}(X)\) \(\text{size}\) returns ones in the “extra” variables, that is, those corresponding to \(\text{ndims}(X)+1\) 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 \( \text{ndims}(X) \):

\[ [d1,d2] = \text{size}(X) \]

\[
\begin{align*}
d1 &= 3 \\
d2 &= 20
\end{align*}
\]
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 serial port object array

Syntax
\[
d = \text{size}(\text{obj}) \\
[m,n] = \text{size}(\text{obj}) \\
[m1,m2,m3,...,mn] = \text{size}(\text{obj}) \\
m = \text{size}(\text{obj},\text{dim})
\]

Arguments
- obj: A serial port object or an array of serial port objects.
- dim: The dimension of obj.
- d: The number of rows and columns in obj.
- m: The number of rows in obj, or the length of the dimension specified by dim.
- n: The number of columns in obj.
- m1,m2,...,mn: The length of the first N dimensions of obj.

Description
- \(d = \text{size}(\text{obj})\) returns the two-element row vector \(d\) containing the number of rows and columns in \(\text{obj}\).
- \([m,n] = \text{size}(\text{obj})\) returns the number of rows and columns in \(\text{obj}\) in separate output variables.
- \([m1,m2,m3,...,mn] = \text{size}(\text{obj})\) returns the length of the first \(n\) dimensions of \(\text{obj}\).
- \(m = \text{size}(\text{obj},\text{dim})\) returns the length of the dimension specified by the scalar \(\text{dim}\). For example, \(\text{size}(\text{obj},1)\) returns the number of rows.

See Also
Functions
- length

2-2843
**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:

```matlab
size(ts.data)
```

If you want the size of each data sample, use the following syntax:

```matlab
getdatasamplesize(ts)
```

**See Also**  
getdatasamplesize, isempty (timeseries), length (timeseries)
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
slice

Purpose

Volumetric slice 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

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')
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

\[ u = xe^{-x^2-y^2-z^2} \]

over the range \(-2 \leq x \leq 2, -2 \leq y \leq 2, -2 \leq z \leq 2: \)

\[ [x,y,z] = \text{meshgrid}(-2:.2:2,-2:.25:2,-2:.16:2); \]
\[ v = x.*\exp(-x.^2-y.^2-z.^2); \]
\[ xslice = [-1.2,.8,2]; yslice = 2; zslice = [-2,0]; \]
\[ \text{slice}(x,y,z,v,xslice,yslice,zslice) \]
\[ \text{colormap hsv} \]
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 1-101 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 $W$.

$W = \text{smooth3}(V, 'filter')$ 
$\text{filter}$ determines the convolution kernel and can be the strings

- 'gaussian'
- 'box' (default)

$W = \text{smooth3}(V, 'filter', \text{size})$ sets the size of the convolution kernel (default is $[3 \ 3 \ 3]$). If $\text{size}$ is scalar, then $\text{size}$ is interpreted as $[\text{size, size, size}]$.

$W = \text{smooth3}(V, 'filter', \text{size}, \text{sd})$ sets an attribute of the convolution kernel. When $\text{filter}$ is gaussian, $\text{sd}$ 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 1-101 for related functions

See Displaying an Isosurface for another example.
Sort array elements in ascending or descending order

Syntax

B = sort(A)
B = sort(A,dim)
B = sort(...,mode)
[B,IX] = sort(A,...)

Description

B = 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., abs(A), and where magnitudes are equal, further sorted by phase angle, i.e., angle(A), on the interval \([-\pi, \pi]\). If A includes any NaN elements, sort places these at the high end.

B = sort(A,dim) sorts the elements along the dimension of A specified by a scalar dim.

B = sort(...,mode) sorts the elements in the specified direction, depending on the value of 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 an 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:

- \( \text{abs}(r) < \text{abs}(s) \)
- \( \text{abs}(r) = \text{abs}(s) \) and \( \text{angle}(r) < \text{angle}(s) \)

where \(-\pi < \text{angle}(r) \leq \pi\)

For example,

```matlab
v = [1 -1 i -i];
angle(v)
an =
    0    3.1416    1.5708   -1.5708
sort(v)
an =
    0    1.0000i   1.0000
    0   -1.0000i  -1.0000
```
sort

Note sort uses a different rule for ordering complex numbers than do max and min, or the relational operators < and >. See the Relational Operators reference page for more information.

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.

\[
A = \begin{bmatrix}
3 & 7 & 5 \\
0 & 4 & 2 \\
\end{bmatrix};
\]

\[
\text{sort}(A,1)
\]

\[
\text{ans} = \\
0 & 4 & 2 \\
3 & 7 & 5 \\
\]

\[
\text{sort}(A,2)
\]

\[
\text{ans} = \\
3 & 5 & 7 \\
0 & 2 & 4 \\
\]

\[
[B,IX] = \text{sort}(A,2)
\]

\[
B = \\
3 & 5 & 7 \\
0 & 2 & 4 \\
\]

\[
IX = \\
1 & 3 & 2 \\
1 & 3 & 2 \\
\]

Example 2

This example sorts each column of a matrix in descending order.
A = [ 3 7 5 
      6 8 3 
      0 4 2 ];

sort(A,1,'descend')

ans =
      6 8 5 
      3 7 3 
      0 4 2

This is equivalent to

sort(A,'descend')

ans =
      6 8 5 
      3 7 3 
      0 4 2

See Also

issorted, max, mean, median, min, sortrows
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, MATLAB 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:

```
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    0   19   41    1
```
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):

```matlab
A = [76, 61, 93, 81; 76, 79, 91, 0; 95, 7, 40, 35; 95, 7, 73, 5; 95, 7, 74, 0];
sortrows(A)
ans =
    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   74   03   56   09   36
```

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 this column are sorted; rows with equal elements in other columns are left in their original order:

```matlab
sortrows(A,1)
ans =
    76   61   93   81   27   46   83
    76   79   91   0   19   41   1
    95   45   92   41   13    1   84
    95    7   73   89   20   74   52
    95    7   73    5   19   44   20
```

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:

```matlab
sortrows(A,[1 7])
ans =
    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:

```
sortrows(A, -4)
```
```
ans =
  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
```

**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`) to the speaker on PC 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 of approximately 5-10 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 supports all Windows-compatible sound devices. Additional sound acquisition and generation capability is available in the Data Acquisition Toolbox. 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`
## soundsc

### Purpose
Scale data and play as sound

### Syntax
- `soundsc(y,Fs)`
- `soundsc(y)`
- `soundsc(y,Fs,bits)`
- `soundsc(y,...,slim)`

### Description
`sounds(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.

`sounds(y)` plays the sound at the default sample rate or 8192 Hz.

`sounds(y,Fs,bits)` plays the sound using `bits` number of bits/sample if possible. Most platforms support `bits = 8` or `bits = 16`.

`sounds(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 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

sparse([],[],[],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**

\[
S = \text{sparse}(A)
\]
\[
S = \text{sparse}(i, j, s, m, n, nzmax)
\]
\[
S = \text{sparse}(i, j, s, m, n)
\]
\[
S = \text{sparse}(i, j, s)
\]
\[
S = \text{sparse}(m, n)
\]

**Description**
The `sparse` function generates matrices in the MATLAB sparse storage organization.

\[
S = \text{sparse}(A)
\]
converts a full matrix to sparse form by squeezing out any zero elements. If \(S\) is already sparse, \(\text{sparse}(S)\) returns \(S\).

\[
S = \text{sparse}(i, j, s, m, n, nzmax)
\]
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.

\[
S = \text{sparse}(i, j, s, m, n)
\]
uses \(nzmax = \text{length}(s)\).

\[
S = \text{sparse}(i, j, s)
\]
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 \([m\ n\ 0]\).

\[
S = \text{sparse}(m, n)
\]
abbreviates \(\text{sparse}([], [], [], m, n, 0)\). 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:

```matlab
[i,j,s] = find(S);
[m,n] = size(S);
S = sparse(i,j,s,m,n);
```

So does this, if the last row and column have nonzero entries:

```matlab
[i,j,s] = find(S);
S = sparse(i,j,s);
```

See Also

diag, find, full, issparse, nnz, nonzeros, nzmax, spones, sprandn, sprandsym, spy

The sparfun directory
**Purpose**
Form least squares augmented system

**Syntax**

```matlab
S = spaugment(A,c)
S = spaugment(A)
```

**Description**

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix

\[
S = \begin{bmatrix} c*I & A \\ A' & 0 \end{bmatrix}
\]

The matrix `S` is related to the least squares problem

\[
\min \| b - A*x \|
\]

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 \( \min(\text{svd}(A)) \) and \( \text{norm}(r) \), which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses \( \max(\max(\text{abs}(A)))/1000 \).

**Note** In previous versions of MATLAB, the augmented matrix was used by sparse linear equation solvers, \ and /, for nonsquare problems. Now, MATLAB 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.

\[ S = \text{spconvert}(D) \]
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 \(\text{spconvert}\) can be used after \(D\) is loaded from either a MAT-file or an ASCII file.

Examples
Suppose the ASCII file uphill.dat contains

\[
\begin{align*}
1 &\ 1 & 1.000000000000000
1 &\ 2 & 0.500000000000000
2 &\ 2 & 0.333333333333333
1 &\ 3 & 0.333333333333333
2 &\ 3 & 0.250000000000000
3 &\ 3 & 0.200000000000000
1 &\ 4 & 0.250000000000000
2 &\ 4 & 0.200000000000000
3 &\ 4 & 0.166666666666667
4 &\ 4 & 0.142857142857143
4 &\ 4 & 0.000000000000000
\end{align*}
\]

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
- \( B = \text{spdiags}(A) \)
- \([B,d] = \text{spdiags}(A)\)
- \( B = \text{spdiags}(A,d) \)
- \( A = \text{spdiags}(B,d,A) \)
- \( A = \text{spdiags}(B,d,m,n) \)

### Description
The `spdiags` function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible.

- \( B = \text{spdiags}(A) \) 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 \( A \).
- \([B,d] = \text{spdiags}(A)\) returns a vector \( d \) of length \( p \), whose integer components specify the diagonals in \( A \).
- \( B = \text{spdiags}(A,d) \) extracts the diagonals specified by \( d \).
- \( A = \text{spdiags}(B,d,A) \) replaces the diagonals specified by \( d \) with the columns of \( B \). The output is sparse.
- \( A = \text{spdiags}(B,d,m,n) \) 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 \( m \geq n \), `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 2-2875 and “Example 5B” on page 2-2877, below).

### Arguments
The `spdiags` function deals with three matrices, in various combinations, as both input and output.
spdiags

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)-by-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

\[
\text{for } k = 1:p \\
\quad B(:,k) = \text{diag}(A,d(k)) \\
\text{end}
\]

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.

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

2-2870
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

\[
B =
\begin{bmatrix}
0 & 0 & 5 & 10 \\
0 & 0 & 6 & 11 \\
0 & 3 & 7 & 12 \\
1 & 4 & 8 & 0 \\
2 & 5 & 9 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
-3 \\
-2 \\
1
\end{bmatrix}
\]
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 2-2870.

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.

\[
e = \text{ones}(n,1);
A = \text{spdiags}([e -2*e e], -1:1, n, n)
\]

Turn it into Wilkinson's test matrix (see gallery):

\[
A = \text{spdiags}(\text{abs}(-(n-1)/2:(n-1)/2)',0,A)
\]

Finally, recover the three diagonals:

\[
B = \text{spdiags}(A)
\]

Example 3

The second example is not square.

\[
A = \begin{bmatrix}
11 & 0 & 13 & 0 \\
0 & 22 & 0 & 24
\end{bmatrix}
\]
Here \( m = 7, n = 4, \) and \( p = 3. \)

The statement \([B,d] = \text{spdiags}(A)\) produces \( d = [-3 \ 0 \ 2]'\) and

\[
B = \begin{bmatrix}
41 & 11 & 0 \\
52 & 22 & 0 \\
63 & 33 & 13 \\
74 & 44 & 24
\end{bmatrix}
\]

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.

\[
B = \text{repmat}((1:6)',[1 \ 7])
\]

\[
B =
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 & 6
\end{bmatrix}
\]

\[
d = [-4 \ -2 \ -1 \ 0 \ 3 \ 4 \ 5];
\]

\[
A = \text{spdiags}(B,d,6,6);
\]

\[
\text{full}(A)
\]

\[
\text{ans} =
\]
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 \( n \).**

\[
A = \text{full}(\texttt{spdiags}(B, [-2 0 2], 5, 5))
\]

<table>
<thead>
<tr>
<th>Matrix B</th>
<th>Matrix A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 6 11</td>
<td>6 0 13 0 0</td>
</tr>
<tr>
<td>2 7 12</td>
<td>0 7 0 14 0</td>
</tr>
<tr>
<td>3 8 13</td>
<td>1 0 8 0 15</td>
</tr>
<tr>
<td>4 9 14</td>
<td>0 2 0 9 0</td>
</tr>
<tr>
<td>5 10 15</td>
<td>0 0 3 0 10</td>
</tr>
</tbody>
</table>

\( A(3,1), A(4,2), \) 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 \( n \).**

\[
A = \text{full}(\texttt{spdiags}(B, [-2 0 2], 5, 4))
\]

<table>
<thead>
<tr>
<th>Matrix B</th>
<th>Matrix A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 6 11</td>
<td>6 0 13 0</td>
</tr>
<tr>
<td>2 7 12</td>
<td>0 7 0 14</td>
</tr>
<tr>
<td>3 8 13</td>
<td>1 0 8 0</td>
</tr>
<tr>
<td>4 9 14</td>
<td>0 2 0 9</td>
</tr>
<tr>
<td>5 10 15</td>
<td>0 0 3 0</td>
</tr>
</tbody>
</table>

Same as in Part A.
Part 3 — m is less than n.

A = full(spdiags(B, [-2 0 2], 4, 5))

Matrix B  Matrix A
1 6 11 6 0 11 0 0
2 7 12 0 7 0 12 0
3 8 13 == spdiags => 3 0 8 0 13
4 9 14 0 4 0 9 0
5 10 15

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 = 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.

Matrix A  Matrix B
6 0 13 0 0 1 6 0
0 7 0 14 0 2 7 0
1 0 8 0 15 == spdiags => 3 8 13
0 2 0 9 0 0 9 14
0 0 3 0 10 0 10 15

Part 2.
Part 3.

Matrix A

\[
\begin{bmatrix}
6 & 0 & 13 & 0 \\
0 & 7 & 0 & 14 \\
1 & 0 & 8 & 0 \\
0 & 2 & 0 & 9 \\
0 & 0 & 3 & 0
\end{bmatrix}
\]

Matrix B

\[
\begin{bmatrix}
1 & 6 & 0 \\
2 & 7 & 0 \\
3 & 8 & 13 \\
0 & 9 & 14 \\
0 & & &
\end{bmatrix}
\]

== spdiags => 3 8 13

See Also
diag
**Purpose**
Calculate specular reflectance

**Syntax**

\[ R = \text{specular}(N_x, N_y, N_z, S, V) \]

**Description**

\[ R = \text{specular}(N_x, N_y, N_z, S, V) \] returns the reflectance of a surface with normal vector components \([N_x, N_y, N_z]\). \(S\) 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 \(\text{specular}(N_x, N_y, N_z, S, V, \text{spread})\).
**Purpose**  
Sparse identity matrix

**Syntax**  
\[ S = \text{speye}(m,n) \]  
\[ S = \text{speye}(n) \]

**Description**  
\[ S = \text{speye}(m,n) \] forms an \( m \)-by-\( n \) sparse matrix with 1s on the main diagonal.  
\[ S = \text{speye}(n) \] abbreviates \( \text{speye}(n,n) \).

**Examples**  
\[ I = \text{speye}(1000) \] 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 \( I = \text{sparse(eye}(1000,1000)) \), 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.

“Parameterizing Functions Called by Function 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

\[ S = spdiags([1:4]',0,4,4) \]

\[
S = \\
(1,1) 1 \\
(2,2) 2 \\
(3,3) 3 \\
(4,4) 4 
\]

Because fun returns nonzero values for all nonzero element of S, f = spfun(@exp,S) has the same sparsity pattern as S.

\[
f = \\
(1,1) 2.7183 \\
(2,2) 7.3891 \\
(3,3) 20.0855 \\
(4,4) 54.5982 
\]
whereas \( \exp(S) \) has 1s where \( S \) has 0s.

\[
\text{full}(\exp(S))
\]

\[
\begin{array}{cccc}
2.7183 & 1.0000 & 1.0000 & 1.0000 \\
1.0000 & 7.3891 & 1.0000 & 1.0000 \\
1.0000 & 1.0000 & 20.0855 & 1.0000 \\
1.0000 & 1.0000 & 1.0000 & 54.5982 \\
\end{array}
\]

**See Also**

`function_handle (@)`
**Purpose**  
Transform spherical coordinates to Cartesian

**Syntax**  
\[ [x,y,z] = \text{sph2cart}(\text{THETA},\text{PHI},R) \]

**Description**  
\[ [x,y,z] = \text{sph2cart}(\text{THETA},\text{PHI},R) \] transforms the corresponding elements of spherical coordinate arrays to Cartesian, or \( xyz \), coordinates. THETA, PHI, and R must all be the same size. 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

**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 1-89 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 $10 \times t$ seconds. The amount of time specified by $t$ depends on your hardware configuration (e.g., if you are running MATLAB over a network).

`spinmap(t,inc)` rotates the colormap for approximately $10 \times t$ 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 1-97 for related functions
Purpose
Cubic spline data interpolation

Syntax
```
pp = spline(x,Y)
yy = spline(x,Y,xx)
```

Description
```
pp = spline(x,Y) returns the piecewise polynomial form of the cubic
spline interpolant for later use with ppval and the spline utility 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
  [d1,d2,...,dk,n], where n is the length of x. The interpolation
  is performed for each d1-by-d2-by-...-dk value in Y. See Exceptions
  (2) for an exception to this rule.
```

```
yy = spline(x,Y,xx) is the same as yy = ppval(spline(x,Y),xx),
thus providing, in yy, the values of the interpolant at xx. xx can be a
scalar, a vector, or a multidimensional array. 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, size(yy) equals [d1, d2, ..., dk, length(xx)].
  - If xx is an array of size [m1,m2,...,mj], size(yy) equals
    [d1,d2,...,dk,m1,m2,...,mj].
```
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 \( Y(:,\ldots,:,j+1) \) for \( j=1:\text{length}(x) \)
- \( \text{Df}(\text{min}(x)) \) matches \( Y(:,:,\ldots,:,1) \)
- \( \text{Df}(\text{max}(x)) \) matches \( Y(:,:,\ldots,:,\text{end}) \)

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.

\[
\begin{align*}
x &= 0:10; \\
y &= \sin(x); \\
xx &= 0:.25:10; \\
yy &= \text{spline}(x,y,xx); \\
\text{plot}(x,y,'o',xx,yy)
\end{align*}
\]
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.

```matlab
x = -4:4;
y = [0 .15 1.12 2.36 2.36 1.46 .49 .06 0];
cs = spline(x,[0 y 0]);
xx = linspace(-4,4,101);
plot(x,y,'o',xx,ppval(cs,xx),'-');
```
Example 3

The two vectors

\[
t = 1900:10:1990;
p = [ 75.995 \hspace{1em} 91.972 \hspace{1em} 105.711 \hspace{1em} 123.203 \hspace{1em} 131.669 \ldots \\
150.697 \hspace{1em} 179.323 \hspace{1em} 203.212 \hspace{1em} 226.505 \hspace{1em} 249.633 ];
\]

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

\[
\text{spline}(t,p,2000)
\]

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];
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.
spline

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**

interpl, ppval, mkpp, pchip, unmkpp

**References**

**Purpose**
Replace nonzero sparse matrix elements with ones

**Syntax**
\[ R = \text{spones}(S) \]

**Description**
\[ R = \text{spones}(S) \] generates a matrix \( R \) with the same sparsity structure as \( S \), but with 1’s in the nonzero positions.

**Examples**
\[ c = \text{sum}(\text{spones}(S)) \] is the number of nonzeros in each column.
\[ r = \text{sum}(\text{spones}(S'))' \] is the number of nonzeros in each row.
\[ \text{sum}(c) \text{ and } \text{sum}(r) \text{ are equal, and are equal to } \text{nnz}(S). \]

**See Also**
\[ \text{nnz}, \text{spalloc}, \text{spfun} \]
**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.
'rreduce' If positive, minimum degree does row reduction every rreduce stages.

'wh_frac' Rows with density > wh_frac are ignored in colmmd.

'autommd' Nonzero to use minimum degree (MMD) orderings with QR-based \ and /.

'autoamd' Nonzero to use colamd ordering with the UMFPACK LU-based \ and /, and to use amd with CHOLMOD Cholesky-based \ and /.

'piv_tol' Pivot tolerance used by the UMFPACK LU-based \ and /.

'bandden' 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.

'umfpack' Nonzero to use UMFPACK instead of the v4 LU-based solver in \ and /.

'sym_tol' Symmetric pivot tolerance used by UMFPACK. See lu for more information about the role of the symmetric pivot tolerance.

**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></th>
<th>Keyword</th>
<th>Default</th>
<th>Tight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'spumoni'</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>'thr_rel'</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>'thr_abs'</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>'exact_d'</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>'supernd'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>'rreduce'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>7</td>
<td>'wh_frac'</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>8</td>
<td>'autommd'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>'autoamd'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>'piv_tol'</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>'bandden'</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>'umfpack'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>'sym_tol'</td>
<td>0.001</td>
<td></td>
</tr>
</tbody>
</table>
Notes

Sparse $A\backslash b$ on Symmetric Positive Definite $A$

Sparse $A\backslash b$ 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 $A\backslash b$ on General Square $A$

Sparse $A\backslash b$ 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 $A\backslash b$ on Rectangular $A$

Sparse $A\backslash b$ 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, colmmd, symmmd

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 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 digit string specifying the minimum number of digits to be printed.</td>
<td>%6f</td>
</tr>
<tr>
<td>Precision</td>
<td>A digit 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>
<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>&quot;r&quot;</td>
<td>Single quotation mark</td>
</tr>
<tr>
<td>(two single quotes)</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>Percent character</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>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 <code>%bx</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>The underlying C data type is a float rather than an unsigned integer.</td>
</tr>
</tbody>
</table>
For example, to print a double value in hexadecimal use the format
'\% bx'.

- The \texttt{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:
  - 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.

\begin{verbatim}
Str = [65 66 67 \pi];
\texttt{sprintf}('\%s \%f', Str)
\texttt{ans} =
ABC 3.141593
\end{verbatim}

- 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).

\begin{verbatim}
Str = [\pi 65 66 67];
\texttt{sprintf}('\%s \%f \%s', Str)
\texttt{ans} =
3.141593e+000 65.000000 BC
\end{verbatim}

- 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 negative zero and exponents differently on some platforms, as shown in the following tables.

### Negative Zero Printed with %e, %E, %f, %g, or %G

<table>
<thead>
<tr>
<th>Platform</th>
<th>%e or %E</th>
<th>%f</th>
<th>%g or %G</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0.000000e+000</td>
<td>0.000000</td>
<td>0</td>
</tr>
<tr>
<td>Others</td>
<td>-0.000000e+00</td>
<td>-0.000000</td>
<td>-0</td>
</tr>
</tbody>
</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><code>sprintf('%0.5g',(1+sqrt(5)))</code></td>
<td>1.618</td>
</tr>
<tr>
<td><code>sprintf('%0.5g',1/eps)</code></td>
<td>4.5036e+15</td>
</tr>
<tr>
<td><code>sprintf('%15.5f',1/eps)</code></td>
<td>4503599627370496.00000</td>
</tr>
<tr>
<td><code>sprintf('%d',round(pi))</code></td>
<td>3</td>
</tr>
</tbody>
</table>
sprintf

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<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, sscanf

**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.

```matlab
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**

\[ X = \text{sqrtm}(A) \]

\[ [X, \text{resnorm}] = \text{sqrtm}(A) \]

\[ [X, \alpha, \text{condest}] = \text{sqrtm}(A) \]

**Description**

\( X = \text{sqrtm}(A) \) is the principal square root of the matrix \( A \), i.e. \( X^2 = A \).

\( X \) 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.

\[ [X, \text{resnorm}] = \text{sqrtm}(A) \] does not print any warning, and returns the residual, \( \frac{\|A - X^2\|_F}{\|A\|_F} \).

\[ [X, \alpha, \text{condest}] = \text{sqrtm}(A) \] returns a stability factor \( \alpha \) and an estimate \( \text{condest} \) of the matrix square root condition number of \( X \). The residual \( \frac{\|A - X^2\|_F}{\|A\|_F} \) is bounded approximately by \( n\alpha\varepsilon \) and the Frobenius norm relative error in \( X \) is bounded approximately by \( n\alpha\varepsilon \text{condest} \), 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 \( X = [0 1; 0 0] \), do not have any square roots, real or complex, and \text{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, \( Y = \text{sqrtnm}(X) \), is a representation of the second difference operator.

\[
Y =
\begin{bmatrix}
2 & -1 & -0 & -0 & -0 \\
-1 & 2 & -1 & 0 & -0 \\
0 & -1 & 2 & -1 & 0 \\
-0 & 0 & -1 & 2 & -1 \\
-0 & -0 & -0 & -1 & 2 \\
\end{bmatrix}
\]

**Example 2**

The matrix

\[
X =
\begin{bmatrix}
7 & 10 \\
15 & 22 \\
\end{bmatrix}
\]

has four square roots. Two of them are

\[
Y_1 =
\begin{bmatrix}
1.5667 & 1.7408 \\
2.6112 & 4.1779 \\
\end{bmatrix}
\]

and

\[
Y_2 =
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
\end{bmatrix}
\]

The other two are \(-Y_1\) and \(-Y_2\). All four can be obtained from the eigenvalues and vectors of \(X\).

\[
[V,D] = \text{eig}(X);
D =
\begin{bmatrix}
0.1386 & 0 \\
0 & 28.8614 \\
\end{bmatrix}
\]
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 $\text{sqrtm}$ function chooses the two plus signs and produces $Y1$, even though $Y2$ is more natural because its entries are integers.

**See Also**

$\text{expm}$, $\text{funm}$, $\text{logm}$
Purpose  Remove singleton dimensions

Syntax  \[ B = \text{squeeze}(A) \]

Description  \[ B = \text{squeeze}(A) \] returns an array \( B \) with the same elements as \( A \), but with all singleton dimensions removed. A singleton dimension is any dimension for which \( \text{size}(A, \text{dim}) = 1 \). Two-dimensional arrays are unaffected by \text{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.

\[
Y = \\
Y(:,:,1) = \\
0.5194 & 0.0346 \\
0.8310 & 0.0535 \\
Y(:,:,2) = \\
Y(:,:,3) = \\
0.5297 \\
0.6711
\]

The command \( Z = \text{squeeze}(Y) \) yields a 2-by-3 matrix:

\[
Z = \\
0.5194 & 0.0346 & 0.5297 \\
0.8310 & 0.0535 & 0.6711
\]

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.

\[
\text{mat} = \\
\text{mat}(\text{:,},1) = \text{mat}(\text{:,},2) = \\
1 & 1
\]
The command `squeeze(mat)` yields a 5-by-1 matrix:

```matlab
squeeze(mat)
```

```
ans =

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

```matlab
size(squeeze(mat))
```

```
ans =

5  1
```

**See Also**
reshape, shiftdim
**Purpose**
Convert state-space filter parameters to transfer function form

**Syntax**
\[ [b,a] = \text{ss2tf}(A,B,C,D,iu) \]

**Description**
ss2tf converts a state-space representation of a given system to an equivalent transfer function representation.

\[ [b,a] = \text{ss2tf}(A,B,C,D,iu) \]
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**

\[
A = \text{sscanf}(s, \text{format}) \\
A = \text{sscanf}(s, \text{format}, \text{size}) \\
[A, \text{count}, \text{errmsg}, \text{nextindex}] = \text{sscanf}(...) 
\]

**Description**

\( A = \text{sscanf}(s, \text{format}) \) reads data from the MATLAB string \( s \), converts it according to the specified \( \text{format} \) string, and returns it in matrix \( A \). \( \text{format} \) is a string specifying the format of the data to be read. See "Remarks" for details. \text{sscanf} is the same as \( \text{fscanf} \) except that it reads the data from a MATLAB string rather than reading it from a file. If \( s \) is a character array with more than one row, \( \text{sscanf} \) reads the characters in column order.

\( A = \text{sscanf}(s, \text{format}, \text{size}) \) reads the amount of data specified by \( \text{size} \) and converts it according to the specified \( \text{format} \) string. \( \text{size} \) 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 ( \text{inf} ), but ( m ) cannot.</td>
</tr>
</tbody>
</table>

Characteristics of the output matrix \( A \) depend on the values read from the input string and on the \( \text{size} \) argument. If \( \text{sscanf} \) reads only numbers, and if \( \text{size} \) is not of the form \( \text{[m,n]} \), matrix \( A \) is a column vector of numbers. If \( \text{sscanf} \) reads only characters or strings, and if \( \text{size} \) is not of the form \( \text{[m,n]} \), matrix \( A \) is a row vector of characters. See the Remarks section for more information.

\( \text{sscanf} \) differs from its C language namesake \( \text{scanf}() \) in an important respect — it is vectorized to return a matrix argument. The \( \text{format} \) string is cycled through the input string until the first of these conditions occurs:

- The \( \text{format} \) string fails to match the data in the input string
- The amount of data specified by size is read
- The end of the string is reached

[A, count, errmsg, nextindex] = sscanf(...) reads data from the MATLAB string (character array) s, converts it according to the specified format string, and returns it in matrix A. count is an optional output argument that returns the number of values successfully read. 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. nextindex is an optional output argument specifying one more than the number of characters scanned in s.

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 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:

```
%12e
```

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 | Signed decimal integer |
| %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` (hexadecimal), `%e` (exponential), `%f` (floating-point), `%g` (general), `%i` (octal), `%o` (octal), and `%x` (hexadecimal). 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. `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 `char(0)`.

**Output Characteristics: Both Numeric and Character Values Read**

When `sscanf` reads a combination of numbers and either characters or strings from the input string, the elements of the output matrix 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 `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 `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 character read from the input.

When the `size` argument is a scalar \( n \), `sscanf` reads at most \( n \) number, character, or string values from the input string. The output matrix contains at most \( n \) rows. `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 `size` argument is a matrix \([m, n]\), `sscanf` reads at most \((m \times n)\) number, character, or string values from the input string. The output matrix contains at most \( m \) rows. `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 2-2923. If `sscanf` reads only characters or strings, see “Output Characteristics: Only Character Values Read” on page 2-2923.

### Examples

#### Example 1

The statements

```plaintext
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:

```plaintext
sscanf('-010', '%i')
ans =
  -8
```

When using `%o`, on the other hand, `sscanf` returns the converted value as unsigned:

```plaintext
sscanf('-010', '%o')
ans =
4.2950e+009
```

#### Example 3

Create character array A representing both character and numeric data:
A = ['abc 46 6 ghi'; 'def 7 89 jkl']
A =
    abc  46  6  ghi
    def  7  89  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 = 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:

for k = 1:2
    C(k,:) = sscanf(A(k, :)', '%*s %d %d %*s', [1, inf]);
end

C
C =
    46  6
    7  89

See Also
eval, sprintf, textread
**Purpose**

Stairstep graph

<table>
<thead>
<tr>
<th><strong>GUI Alternatives</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

**Syntax**

```matlab
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(...,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.

**Examples**

Create a stairstep plot of a sine wave.

```plaintext
x = linspace(-2*pi,2*pi,40);
stairs(x,sin(x))
```
See Also

bar, hist, stem

“Discrete Data Plots” on page 1-88 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.

- **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.
Stairseries Properties

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.

**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

_Label used by plot legends._ The `legend` function, the figure’s active legend, and the plot browser use this text when displaying labels for this object.

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.

**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.

**LineStyle**

{\texttt{-} | \texttt{--} | \texttt{:} | \texttt{-.} | \texttt{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>
**Stairseries Properties**

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.
Stairseries Properties

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 = 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.
Stairseries Properties

```
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 stairseries objects, Type is 'hggroup'. The following statement finds all the hggroup objects in the current axes object.

```
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 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, \text{length}(XData) == \text{size}(YData,1).

If you do not specify XData (i.e., the input argument \text{x}), the stairs function uses the indices of YData to create the staiirstep graph. 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 \text{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
Stairseries Properties

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 \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.
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**
MATLAB 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 MATLAB 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 MATLAB search path.

You can create a startup.m file in your own MATLAB startup directory. The file can include physical constants, Handle Graphics defaults, engineering conversion factors, or anything else you want predefined in your workspace.

There are other ways to redefine aspects of MATLAB. See Startup Options and About Preferences in the MATLAB Desktop Tools and Development Environment documentation.

**Algorithm**
Only matlabrc.m is actually invoked by MATLAB at startup. However, matlabrc.m contains the statements

```matlab
if exist('startup')==2
    startup
end
```

that invoke startup.m. You can extend this process to create additional startup M-files, if required.

**See Also**
matlabrc, matlabroot, path, quit
**Purpose**
Standard deviation

**Syntax**
\[
s = \text{std}(X)
\]
\[
s = \text{std}(X,\text{flag})
\]
\[
s = \text{std}(X,\text{flag, dim})
\]

**Definition**
There are two common textbook definitions for the standard deviation \( s \) of a data vector \( X \).

\[
(1) \quad s = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2\right)^{\frac{1}{2}}
\]

\[
(2) \quad s = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2\right)^{\frac{1}{2}}
\]

where

\[
\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i
\]

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 =
   1   5   9
   7  15  22
s = std(X,0,1)
s =
   4.2426  7.0711  9.1924
s = std(X,0,2)
s =
   4.000
   7.5056
```

**See Also**
corrcoef, cov, mean, median, var
**Purpose**

Standard deviation of timeseries data

**Syntax**

\[
\text{ts}\_\text{std} = \text{std}(\text{ts}) \\
\text{ts}\_\text{std} = \text{std}(\text{ts}, '\text{PropertyName1}', \text{PropertyValue1}, \ldots)
\]

**Description**

\(\text{ts}\_\text{std} = \text{std}(\text{ts})\) returns the standard deviation of the time-series data. When \(\text{ts}.\text{Data}\) is a vector, \(\text{ts}\_\text{std}\) is the standard deviation of \(\text{ts}.\text{Data}\) values. When \(\text{ts}.\text{Data}\) is a matrix, \(\text{ts}\_\text{std}\) is the standard deviation of each column of \(\text{ts}.\text{Data}\) (when \(\text{IsTimeFirst}\) is true and the first dimension of \(\text{ts}\) is aligned with time). For the N-dimensional \(\text{ts}.\text{Data}\) array, \text{std} always operates along the first nonsingleton dimension of \(\text{ts}.\text{Data}\).

\(\text{ts}\_\text{std} = \text{std}(\text{ts}, '\text{PropertyName1}', \text{PropertyValue1}, \ldots)\) 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.

   ```matlab
   load count.dat
   ```

2. Create a timeseries object with 24 time values.

   ```matlab
   count_ts = timeseries(count, 1:24, 'Name', 'CountPerSecond')
   ```
Calculate the standard deviation of each data column for this timeseries object.

```
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
stem

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(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

See Plot Objects and Backward Compatibility for more information.

**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.
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 \( \exp(-.07\times x) \times \cos(x) \).
- `h(2)` is the handle to the stemseries object plotting the expression \( \exp(.05\times x) \times \cos(x) \).

```matlab
x = 0:25;
y = [exp(-.07*x).*cos(x);exp(.05*x).*cos(x)]';
h = stem(x,y);
set(h(1),'MarkerFaceColor','blue')
set(h(2),'MarkerFaceColor','red','Marker','square')
```
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
stem3

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

stem3(Z)
stem3(X,Y,Z)
stem3(...,'fill')
stem3(...,LineSpec)
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.

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 1-88 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.

**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 MATLAB 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
Stemseries Properties

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.

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
Stemseries Properties

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

Label used by plot legends. The legend function, the figure’s active legend, and the plot browser use this text when displaying labels for this object.

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.

---

**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.

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

Line style. This property specifies the line style of the object. Available line styles are shown in the following table.
### Stemseries Properties

<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>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
</tbody>
</table>
### Stemseries Properties

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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
```

*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, 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

*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:

```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, \( \text{length}(\text{XData}) == \text{size}(\text{YData},1) \). 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)

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
**Stemseries Properties**

*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)

*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)

Arguments
obj  A serial port object or an array of serial port objects.

Description
stopasync(obj) stops any asynchronous read or write operation that is in progress for 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

\[
X = \text{str2double}(\text{'str'}) \\
X = \text{str2double}(C)
\]

Description

\( X = \text{str2double}(\text{'str'}) \) converts the string \( \text{str} \), 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 \( \text{str} \) does not represent a valid scalar value, \( \text{str2double} \) returns NaN.

\( X = \text{str2double}(C) \) 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 \( \text{str2double} \) conversions.

\[
\begin{align*}
\text{str2double}('123.45e7') \\
\text{str2double}('123 + 45i') \\
\text{str2double('3.14159')} \\
\text{str2double('2.7i - 3.14')} \\
\text{str2double({'2.71' '3.1415'})} \\
\text{str2double('1,200.34')} \\
\end{align*}
\]

See Also

char, hex2num, num2str, str2num
Purpose
Construct function handle from function name string

Syntax
str2func('str')

Description
str2func('str') constructs a function handle fhandle for the function named in the string 'str'.

You can create a function handle using either the @function syntax or the str2func command. 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.

Examples
Example 1
To convert the string, 'sin', into a handle for that function, type

    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:

    function fh = makeHandle(funcname)
    fh = str2func(funcname);

This is the code that calls makeHandle to construct the function handle:

    makeHandle('sin')
    ans =
         @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)
ans =
    0.2837
```

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
fminbnd(fhandle, lower, upper)
```

Whether you call `myminbnd` with a function handle or function name string, the function can handle the argument appropriately:

```matlab
myminbnd('humps', 0.3, 1)
converting function string to function handle ...
ans =
    0.6370
```

See Also

`function_handle`, `func2str`, `functions`
**Purpose**
Form blank-padded character matrix from strings

**Syntax**

\[
S = \text{str2mat}(T_1, T_2, T_3, \ldots)
\]

**Description**

\[
S = \text{str2mat}(T_1, T_2, T_3, \ldots)
\]
forms the matrix \( S \) containing the text strings \( T_1, T_2, T_3, \ldots \) as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, \( T_i \), 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**

\[
x = \text{str2mat}('36842', '39751', '38453', '90307');
\]

whos x

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4x5</td>
<td>40</td>
<td>char array</td>
</tr>
</tbody>
</table>

\[
x(2,3)
\]

ans =

7

**See Also**

char, strvcat
Purpose

Convert string to number

Syntax

$x = \text{str2num}('str')$

$[x \text{ status}] = \text{str2num}('str')$

Description

$x = \text{str2num}('str')$ converts the string $str$, which is an ASCII character representation of a numeric value, to numeric representation. $\text{str2num}$ also converts string matrices to numeric matrices. If the input string does not represent a valid number or matrix, $\text{str2num}(str)$ returns the empty matrix in $x$.

The input string can contain

- Digits
- A decimal point
- A leading + or - sign
- A letter $e$ or $d$ preceding a power of 10 scale factor
- A letter $i$ or $j$ indicating a complex or imaginary number.

$[x \text{ status}] = \text{str2num}('str')$ returns the status of the conversion in logical status, where status equals logical 1 (true) if the conversion succeeds, and logical 0 (false) otherwise. If the input string $str$ does not represent a valid number or matrix, MATLAB sets $x$ to the empty matrix. If the conversions fails, status is set to 0.

Space characters can be significant. For instance, $\text{str2num}('1+2i')$ and $\text{str2num}('1 + 2i')$ produce $x = 1+2i$, while $\text{str2num}('1 +2i')$ produces $x = [1 2i]$. You can avoid these problems by using the $\text{str2double}$ function.

Note $\text{str2num}$ uses the eval function to convert the input argument, so side effects can occur if the string contains calls to functions. Use $\text{str2double}$ to avoid such side effects or when $S$ contains a single number.
**Examples**

```
str2num('3.14159e0') is approximately π.
```

To convert a string matrix,

```
str2num([ '1 2'; '3 4' ])  
```

```
ans =

    1   2
    3   4
```

**See Also**

num2str, hex2num, sscanf, sparse, special characters
Purpose
Concateenate 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.

Trailing spaces in character array inputs are ignored and do not appear in the output. This is not true for inputs that are cell arrays of strings. Use the concatenation syntax [s1 s2 s3 ...] to preserve trailing spaces.

Remarks
strcat and matrix operation are different for strings that contain trailing spaces:

```plaintext
a = 'hello '
b = 'goodbye'
strcat(a, b)
ans =
hellogoodbye
[a b]
ans =
hello goodbye
```

Examples
Given two 1-by-2 cell arrays a and b,

```plaintext
a = 'abcde'
b = 'fghi'
'jkl' 'mn'
```

the command t = strcat(a,b) yields

```plaintext
2-2989
```
t =
'abcdejkl'   'fghimn'

Given the 1-by-1 cell array c = {`Q'}, the command t = strcat(a,b,c) yields

t =
'abcdejklQ'   'fghimnQ'

See Also
strvcat, cat, cellstr
Purpose

Compare strings

Syntax

```
TF = strcmp('str1', 'str2')
TF = strcmp('str', C)
TF = strcmp(C1, C2)
```

Each of these syntaxes apply 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:

```
TF = strcmp('str1', 'str2') compares the strings str1 and str2 and returns logical 1 (true) if they are identical, and returns logical 0 (false) otherwise.

TF = strcmp('str', C) compares string str to the 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 = strcmp(C1, C2) compares each element of C1 to the same element in C2, where C1 and C2 are equal-size cell arrays of strings. Input C1 and/or C2 can also be a character array with the right number of rows. 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 strcmp and strcmpi is not the same as the C language convention.
**strcmp, strcmpi**

strcmp and strcmpi support international character sets.

**Examples**

Perform a simple comparison of two strings:

```matlab
strcmp('Yes', 'No')
ans =
    0
strcmp('Yes', 'Yes')
ans =
    1
```

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'};
```

Perform a comparison of two cell arrays of strings. 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}:

```matlab
strcmpi(B, C)
ans =
   1    0
   0    1
```
See Also

strcmp, strncmp, strncmpi, strcmpi, strmatch, strfind, findstr, regexp, regexpi, regexpprep, 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:

   [stepsiz]  

or

   [stepsiz, max_number_vertices]

If you do not specify a value, MATLAB 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 1-101 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 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 1-101 for related functions

Specifying Starting Points for Stream Plots for related information
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)
streamline(axes_handle,...)
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 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 1-101 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

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 \( n = 1 \).

`streamparticles(...,'PropertyName',PropertyValue,...)` 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. `Inf` 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. `Inf`, 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 `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`. `streamparticles` sets the following line properties when called.

<table>
<thead>
<tr>
<th>Line Property</th>
<th>Value Set by <code>streamparticles</code></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>
Line Property | Value Set by `streamparticles`
---|---
MarkerEdgeColor | none
MarkerFaceColor | red

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:

```matlab
streamparticles(vertices,'MarkerFaceColor', [0.5 0.5 0.5])
```

`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.

```matlab
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = 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 1-101 for related functions

Creating Stream Particle Animations for more details

Specifying Starting Points for Stream Plots for related information
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,...)`
- `h = streamribbon(...)`

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. 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 twistangle for the twist of the ribbons (in radians). The size of each corresponding element of vertices and twistangle must be equal.

\text{streamribbon}(\ldots,\text{width})\) sets the width of the ribbons to \text{width}.

\text{streamribbon}(\text{axes}\_\text{handle},\ldots)\) plots into the axes object with the handle \text{axes}\_\text{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,sy,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.
streamribbon

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.
%-----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,sy,sz);
[sx sy sz] = meshgrid([1:6],[0],-6);
streamribbon(x,y,z,u,v,w,sx,sy,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 1-101 for related functions
Displaying Curl with Stream Ribbons for another example
Specifying Starting Points for Stream Plots for related information
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)
• 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 `streamslice` to draw lines that indicate the gradient of the surface. `interp2` is used to find the points for the lines that lie on the surface.

```matlab
z = peaks;
surf(z)
shading interp
hold on
```
[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
streamslice

See Also

contourslice, slice, streamline, volumebounds

“Volume Visualization” on page 1-101 for related functions

Specifying Starting Points for Stream Plots for related information
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])
h = streamtube(...)

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

2-3021
streamtube

$$[X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)$$

where $$[m,n,p] = \text{size}(U)$$.

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 $$[m,n,p] = \text{size}(\text{divergence})$$.

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.

```matlab
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
```
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
tooltip
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
```
See Also
divergence, streamribbon, streamline, stream3
“Volume Visualization” on page 1-101 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 \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 \text{S}:

\[
\begin{align*}
\text{S} & = '\text{Find the starting indices of the pattern string}' \\
\text{strfind}(\text{S}, 'in') & \\
\text{ans} & = \\
& \begin{bmatrix} 2 & 15 & 19 & 45 \end{bmatrix} \\
\text{strfind}(\text{S}, 'In') & \\
\text{ans} & = \\
& [] \\
\text{strfind}(\text{S}, ' ') & \\
\text{ans} & = \\
& \begin{bmatrix} 5 & 9 & 18 & 26 & 29 & 33 & 41 \end{bmatrix}
\end{align*}
\]

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{:, :}
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, strcmpl, strncmpi, regexp, regexpi, regexprep
**Purpose**
MATLAB string handling

**Syntax**
- \( S = '\text{Any Characters}' \)
- \( S = [S_1 \ S_2 \ldots] \)
- \( S = \text{strcat}(S_1, \ S_2, \ldots) \)

**Description**

- \( S = '\text{Any Characters}' \) 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.

- \( S = [S_1 \ S_2 \ldots] \) concatenates character arrays \( S_1, S_2, \) etc. into a new character array, \( S \).

- \( S = \text{strcat}(S_1, \ S_2, \ldots) \) concatenates \( S_1, S_2, \) 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, \( \text{strcat} \) returns a cell array of strings.

Trailing spaces in \( \text{strcat} \) character array inputs are ignored and do not appear in the output. This is not true for \( \text{strcat} \) inputs that are cell arrays of strings. Use the \( S = [S_1 \ S_2 \ldots] \) concatenation syntax, shown above, to preserve trailing spaces.

- \( S = \text{char}(X) \) can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.

- \( X = \text{double}(S) \) 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 \( \text{strvcat} \)
- As a cell array of strings via the curly braces

You can convert between character array and cell array of strings using \( \text{char} \) and \( \text{cellstr} \). 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!'
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')
C =
Hello
Yes
No
Goodbye
```

Create a cell array of strings.

```matlab
S = {'Hello' 'Yes' 'No' 'Goodbye'}
S =
'Hello' 'Yes' 'No' 'Goodbye'
```

**See Also**

char, isstrprop, cellstr, ischar, isletter, isspace, iscellstr, strvcat, sprintf, sscanf, text, input
Purpose

Justify character array

Syntax

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

deblank, strtrim
**Purpose**
Find possible matches for string

**Syntax**

- \( x = \text{strmatch}(str, \text{strarray}) \)
- \( x = \text{strmatch}(str, \text{strarray}, \text{'exact'}) \)

**Description**

- \( x = \text{strmatch}(str, \text{strarray}) \) looks through the rows of the character array or cell array of strings \(\text{strarray} \) to find strings that begin with the text contained in \( str \), and returns the matching row indices. Any trailing space characters in \( str \) or \( \text{strarray} \) are ignored when matching. \(\text{strmatch} \) is fastest when \( \text{strarray} \) is a character array.

- \( x = \text{strmatch}(\text{str}, \text{strarray}, \text{'exact'}) \) compares \( \text{str} \) with each row of \( \text{strarray} \), looking for an exact match of the entire strings. Any trailing space characters in \( \text{str} \) or \( \text{strarray} \) are ignored when matching.

**Examples**
The statement

\[
 x = \text{strmatch}'max', \text{strvcat}'max', \text{strvcat}'minimax', \text{strvcat}'maximum'
\]

returns \( x = [1; 3] \) since rows 1 and 3 begin with 'max'. The statement

\[
 x = \text{strmatch}'max', \text{strvcat}'max', \text{strvcat}'minimax', \text{strvcat}'maximum', \text{'exact'}
\]

returns \( x = 1 \), since only row 1 matches 'max' exactly.

**See Also**
\( \text{strcmp}, \text{strcmpi}, \text{strncmp}, \text{strncmpi}, \text{strfind}, \text{findstr}, \text{strvcat}, \text{regexp}, \text{regexpi}, \text{regexprep} \)
strncmp, strncmpi

**Purpose**

Compare first \( n \) characters of strings

**Syntax**

\[
\begin{align*}
\text{TF} &= \text{strncmp('str1', 'str2', n)} \\
\text{TF} &= \text{strncmp('str', C, n)} \\
\text{TF} &= \text{strncmp(C1, C2, n)}
\end{align*}
\]

Each of these syntaxes apply to both \text{strncmp} and \text{strncmpi}. The \text{strncmp} function is case sensitive in matching strings, while \text{strncmpi} is not:

**Description**

Although the following descriptions show only \text{strncmp}, they apply to \text{strncmpi} as well. The two functions are the same except that \text{strncmpi} compares strings without sensitivity to letter case:

\[
\begin{align*}
\text{TF} &= \text{strncmp('str1', 'str2', n)} \text{ compares the first } n \text{ characters of strings str1 and str2 and returns logical 1 (true) if they are identical, and returns logical 0 (false) otherwise.}
\end{align*}
\]

\[
\begin{align*}
\text{TF} &= \text{strncmp('str', C, n)} \text{ compares the first } n \text{ characters of str to the first } n \text{ characters of each element of cell array C, where str is a character vector (or a 1-by-1 cell array), and C is a cell array of strings. The function returns TF, a logical array that is the same size as C and contains logical 1 (true) for those elements of C that are a match, and logical 0 (false) for those elements that are not. The order of the first two input arguments is not important.}
\end{align*}
\]

\[
\begin{align*}
\text{TF} &= \text{strncmp(C1, C2, n)} \text{ compares each element of C1 to the same element in C2, where C1 and C2 are equal-size cell arrays of strings. Input C1 and/or C2 can also be a character array with the right number of rows. The function attempts to match only the first } n \text{ characters of each string. The function returns TF, a logical array that is the same size as C1 and C2, and contains logical 1 (true) for those elements of C1 and C2 that are a match, and logical 0 (false) for those elements that are not.}
\end{align*}
\]

**Remarks**

These functions are intended for comparison of character data. When used to compare numeric data, they return logical 0.
strncmp, strncmpi

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

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
```

See Also

strcmp, strc mpi, strmatch, strfind, findstr, regexp, reg expi, regexprep, regextranslate
**Purpose**

Read formatted data from string

---

**Note**

The `textscan` function is intended as a replacement for both `strread` and `textread`.

---

**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 2-3038 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 2-3038 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 2-3039 and “Example 5” on page 2-3039 below.

The table `Formats for strread` on page 2-3035 lists the valid format specifiers. More information on using formats is available under “Formats” on page 2-3037 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 2-3039 below.

[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 2-3036 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 2-3040 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 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 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>
</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>&quot;* where * can be \b\f\n\r\t &quot; or &quot;%&quot;</td>
<td>Treats vector of characters, *, as white space. Default is \b\f\n\r\t.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Backspace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Form feed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New line</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Carriage return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Horizontal tab</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Backslash</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single quotation mark</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Percent sign</td>
</tr>
<tr>
<td>param</td>
<td>value</td>
<td>Action</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</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>expchar</td>
<td>Exponent characters</td>
<td>Default is eEdD.</td>
</tr>
<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></td>
<td>shell</td>
<td>Ignores characters after #.</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>Ignores characters between /* and */.</td>
</tr>
<tr>
<td></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 `strread` parameter `delimiter` to specify the delimiter. For example, if the string `str` used a semicolon as a delimiter, you would use this command:

```matlab
[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 `strread` function continues reading `str` until the entire string is read. If there are fewer format specifiers than there are entities in `str`, `strread` reapplications the format specifiers, starting over at the beginning. See “Example 5” on page 2-3039 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:

```
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:

```
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:

```
[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:

\[
a = \text{strread('0.41 8.24 3.57 6.24 9.27', '%%4.2f', 3)}
\]

\[
a =
\begin{align*}
0.4100 \\
8.2400 \\
3.5700
\end{align*}
\]

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:

\[
a = \text{strread('0.41 8.24 3.57 6.24 9.27', '%%3.1f %*1d')}
\]

\[
a =
\begin{align*}
0.4000 \\
8.2000 \\
3.5000 \\
6.2000 \\
9.2000
\end{align*}
\]

Example 5

Read six numbers into two variables, reusing the format specifiers:

\[
[a ~ b] = \text{strread('0.41 8.24 3.57 6.24 9.27 3.29', '%%f %%f')}
\]

\[
a =
\begin{align*}
0.4100 \\
3.5700 \\
9.2700
\end{align*}
\]

\[
b =
\begin{align*}
8.2400 \\
6.2400
\end{align*}
\]
strread

Example 6

Read string and numeric data to two output variables. Ignore commas in the input string:

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:

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:

str = '<table border=5 width="100%" cellspacing=0>,'

[border width space] = strread(str, ...
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:

```matlab
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, textread, 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
strtok

**Purpose**
Selected parts of string

**Syntax**

```matlab
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 MATLAB 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) [4]` 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:

```matlab
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:

```
s = sprintf('%s%s%s', ... 
'\<ul class=continued>\<li class=continued\\>', ... 
'\<pre><a name="13474"></a>token = strtok', ... 
'\"\<\,' 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
/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:

```matlab
token =
    'all'
    'my'
    'leave'
token =
    'in'
    'dog'
    'no'
token =
    'good'
    'has'
    'stone'
token =
    'time'
    'fleas'
    'unturned'
```

See Also

`findstr`, `strmatch`
strtrim

**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**

```plaintext
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 {}, MATLAB 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.

```plaintext
s = struct('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 identical to the underlying structure in the object obj. The class information is lost.
```
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])
```

```matlab
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
```matlab
s = struct('type', {'big','little'}, 'color', {'red'}, ...
'x', {3 4})
```

produces a structure array `s`:

```
s =  
1x2 struct array with fields: 
  type
  color
  x
```

The value arrays have been distributed among the fields of `s`:

```
s(1)
as = 
  type: 'big'
  color: 'red'
  x: 3
s(2)
as = 
  type: 'little'
  color: 'red'
  x: 4
```

**Example 2**

Similarly, the command

```
a.b = struct('z', {});
```

produces an empty structure `a.b` with field `z`.

```
a.b
ans = 
  0x0 struct array with fields: 
    z
```
Example 3

This example initializes one field \( f_1 \) using a cell array, and the other \( f_2 \) using a scalar value:

```matlab
s = struct('f1', {1 3; 2 4}, 'f2', 25)
s =
2x2 struct array with fields:
f1
    f2
Field \( f_1 \) 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 \( f_2 \) 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
\[ c = \text{struct2cell}(s) \]

Description
\[ c = \text{struct2cell}(s) \] 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 \([p \ \text{size}(s)]\).

Examples
The commands
\[
\begin{align*}
\text{clear } s, & \quad s.\text{category} = 'tree'; \\
s.\text{height} = 37.4; & \quad s.\text{name} = 'birch';
\end{align*}
\]
create the structure
\[
s = 
\begin{align*}
\text{category}: & \quad 'tree' \\
\text{height}: & \quad 37.4000 \\
\text{name}: & \quad 'birch'
\end{align*}
\]
Converting the structure to a cell array,
\[
c = \text{struct2cell}(s)
\]
\[
c = 
\begin{align*}
'\text{tree}' \\
[37.4000] \\
'\text{birch}'
\end{align*}
\]

See Also
cell2struct, cell, iscell, struct, isstruct, fieldnames, “Using Dynamic Field Names”
Purpose
Apply function to each field of scalar structure

Syntax
A = structfun(fun, S)
[A, B, ...] = structfun(fun, S)
[A, ...] = structfun(fun, S, 'param1', value1, ...)

Description
A = structfun(fun, S) applies the function specified by fun to each
field of scalar structure S, and returns the results in array A. 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 Nth element of A is the
result of applying fun to the Nth field of S, and the order of the fields is
the same as that returned by a call to fieldnames.

fun must return values of the same class each time it is called. If fun is
a handle to an overloaded function, then structfun follows MATLAB
dispatching rules in calling the function.

[A, B, ...] = structfun(fun, S) returns arrays A, B, ..., each
array corresponding to one of the output arguments of fun. structfun
calls fun each time with as many outputs as there are in the call to
structfun. fun can return output arguments having different classes,
but the class of each output must be the same each time fun is called.

[A, ...] = structfun(fun, S, 'param1', value1, ...) enables
you to specify optional parameter name/parameter value pairs.
Parameters are
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'UniformOutput'</td>
<td>Logical value indicating whether or not the outputs of ( \text{fun} ) can be returned without encapsulation in a structure. The default value is true.</td>
</tr>
<tr>
<td></td>
<td>If equal to logical 1 (true), ( \text{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.</td>
</tr>
<tr>
<td></td>
<td>If equal to logical 0 (false), ( \text{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 ( \text{fun} ) on the corresponding values in the input structure ( B ). In this case, the outputs can be of any data type.</td>
</tr>
<tr>
<td>'ErrorHandler'</td>
<td>Function handle specifying the function MATLAB is to call if the call to ( \text{fun} ) fails. MATLAB calls the error handling function with the following input arguments:</td>
</tr>
<tr>
<td></td>
<td>• 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 field names) at which the error occurred.</td>
</tr>
<tr>
<td></td>
<td>• The input argument at which the call to the function failed.</td>
</tr>
<tr>
<td></td>
<td>The error handling function should either rethrow an error or return the same number of outputs as ( \text{fun} ). These outputs are then returned as the outputs of ( \text{structfun} ). If 'UniformOutput' is true, the outputs of the error handler must also be scalars of the same type as the outputs of ( \text{fun} ).</td>
</tr>
<tr>
<td></td>
<td>For example,</td>
</tr>
</tbody>
</table>
|                 | \[
|                 | \begin{verbatim}
|                 | \text{function [A, B] = errorFunc(S, ...}
|                 | \text{varargin)}
|                 | \text{warning(S.identifier, S.message);}
|                 | \text{A = NaN; B = NaN;}
|                 | \end{verbatim}
|                 | \]                                                                                                                                                                                                    |
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';

shortNames = structfun(@(x) ( x(1:3) ), s, ...
    'UniformOutput', false);
```

See Also
```matlab
cellfun, arrayfun, function_handle, cell2mat, spfun```

**Purpose**
Concatenate strings vertically

**Syntax**

\[
S = \text{strvcat}(t_1, t_2, t_3, \ldots) \\
S = \text{strvcat}(c)
\]

**Description**

\[S = \text{strvcat}(t_1, t_2, t_3, \ldots)\] forms the character array \(S\) containing the text strings (or string matrices) \(t_1, t_2, t_3, \ldots\) as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

\[S = \text{strvcat}(c)\] when \(c\) is a cell array of strings, passes each element of \(c\) as an input to \text{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.

\[
t_1 = 'first'; t_2 = 'string'; t_3 = 'matrix'; t_4 = 'second'; \\
S1 = \text{strvcat}(t_1, t_2, t_3) \quad S2 = \text{strvcat}(t_4, t_2, t_3)
\]

\[
S1 = \\
\begin{array}{c}
\text{first} \\
\text{string} \\
\text{matrix}
\end{array} \\
S2 = \\
\begin{array}{c}
\text{second} \\
\text{string} \\
\text{matrix}
\end{array}
\]

\[
S3 = \text{strvcat}(S1, S2)
\]

\[
S3 = \\
\begin{array}{c}
\text{first} \\
\text{string} \\
\text{matrix}
\end{array}
\]

\[
S3 = \\
\begin{array}{c}
\text{second} \\
\text{string}
\end{array}
\]
matrix

See Also  strcat, cat, int2str, mat2str, num2str, strings
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 2-element vector, 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.

sub2ind(size(A),2,1,2)

ans =

14

You can now access the same location in A using the single subscripting method.

A(14)

ans =

-8

See Also

ind2sub, find, size
Purpose

Create axes in tiled positions

GUI

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,'v6')
subplot(h)
subplot('Position', [left bottom width height])
h = subplot(...)

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 outgo 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.

If p is a vector, it specifies an axes object having a position that covers all the subplot positions listed in p.

```
 subplot(m,n,p,'replace')  
```

If the specified axes object already exists, delete it and create a new axes.

```
 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.

```
 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.

```
 h = subplot(...)  
```

returns the handle to the new axes object.

**Backwards Compatibility**

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.

**Remarks**

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

```
get(h(i), '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**

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)
subplot(2,1,2); plot(outgo)
```
The following illustration shows four subplot regions and indicates the command used to create each.
The following combinations produce asymmetrical arrangements of subplots.

```
subplot(2,2,[1 3])
subplot(2,2,2)
subplot(2,2,4)
```
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)
```
See Also

axes, cla, clf, figure, gca

“Basic Plots and Graphs” on page 1-85 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.

**Remarks**
\texttt{subsasgn} is designed to be used by the MATLAB interpreter to handle indexed assignments to objects. Calling \texttt{subsasgn} directly as a function is not recommended. If you do use \texttt{subsasgn} in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.

In the assignment \( A(J,K,\ldots)=B(M,N,\ldots) \), subscripts \( J,K,M,N,\ldots \) may be scalar, vector, or array, provided that all of the following are true:

- The number of subscripts specified for \( B \), excluding trailing subscripts equal to 1, does not exceed \texttt{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 \texttt{numel} reference page for information concerning the use of \texttt{numel} with regards to the overloaded \texttt{subsasgn} function.
If A is an array of one of the fundamental MATLAB data types, then assigning a value to A with indexed assignment calls the builtin MATLAB subsasgn method. It does not call any subsasgn method that you may have overloaded for that data type. 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 builtin 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

See “Handling Subscripted Assignment” for more information about overloaded methods and subsasgn.
Purpose

Subscripted indexing for objects

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 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.

\[H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);\]

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.

\[theta = subspace(A,B)\]
\[theta = 1.5708\]

That A and B are orthogonal is shown by the fact that theta is equal to \(\pi/2\).

\[theta - pi/2\]
\[ans = 0\]
Purpose
Subscripted reference for objects

Syntax
B = subsref(A, S)

Description
B = subsref(A, S) is called for the syntax A(i), A{i}, 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 fundamental MATLAB data types, then
referencing a value of A using an indexed reference calls the builtin
MATLAB subsref method. It does not call any subsref method that
you may have overloaded for that data type. 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 builtin subsref method instead.

Examples
The syntax A(1:2,:) calls subsref(A,S) 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} calls subsref(A,S) where S.type='{}' and
S.subs={1:2}.

The syntax A.field calls subsref(A,S) 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)` calls `subsref(A,S)` 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**

`subsasgn`, `substruct`

See “Handling Subscripted Reference” for more information about overloaded methods and `subsref`. 
**Purpose**
Create structure argument for subsasgn or subsref

**Syntax**

\[ S = \text{substruct}(\text{type1}, \text{subs1}, \text{type2}, \text{subs2}, \ldots) \]

**Description**

\[ S = \text{substruct}(\text{type1}, \text{subs1}, \text{type2}, \text{subs2}, \ldots) \] 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

\[ B = \text{A}(3,5).\text{field} \]

you can use

\[
\begin{align*}
S &= \text{substruct}('()', \{3,5\}, '.', 'field'); \\
B &= \text{subsref}(\text{A}, S);
\end{align*}
\]

The structure created by substruct in this example contains the following:

\[ S(1) \]

\[
\text{ans} = \\
\begin{cases}
\text{type}: '()' \\
\text{subs}: \{[3] \ [5]\}
\end{cases}
\]

\[ S(2) \]
ans =
    type: '.'
    subs: 'field'

See Also
      subsasgn, subsref
Purpose
Extract subset of volume data set

Syntax
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)
[Nx,Ny,Nz,Nv] = subvolume(V,limits)
Nv = subvolume(...)

Description
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits) extracts a subset of
the volume data set V using the specified axis-aligned limits. limits =
[xmin,xmax,ymin, ymax,zmin,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] = subvolume(V,limits) 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(V).
Nv = subvolume(...) 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 1-101 for related functions
**Purpose**
Sum of array elements

**Syntax**

B = sum(A)
B = sum(A,dim)
B = sum(..., 'double')
B = sum(..., dim,'double')
B = sum(..., 'native')
B = sum(..., dim,'native')

**Description**

B = sum(A) returns sums along different dimensions of an array.

If A is a vector, sum(A) returns the sum of the elements.

If A is a matrix, 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, sum(A) treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

B = sum(A,dim) sums along the dimension of A specified by scalar dim. The dim input is an integer value from 1 to N, where N is the number of dimensions in A. Set dim to 1 to compute the sum of each column, 2 to sum rows, etc.

B = sum(..., 'double') and B = sum(..., dim,'double') performs additions in double-precision and return an answer of type double, even if A has data type single or an integer data type. This is the default for integer data types.

B = sum(..., 'native') and B = sum(..., dim,'native') performs additions in the native data type of A and return an answer of the same data type. This is the default for single and double.

**Remarks**

sum(diag(X)) is the trace of X.

**Examples**
The magic square of order 3 is

```matlab
M = magic(3)
M =
```
This is called a magic square because the sums of the elements in each column are the same.

\[
\begin{array}{ccc}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2 \\
\end{array}
\]

\[\text{sum}(M) = \]
\[
\begin{array}{ccc}
15 & 15 & 15 \\
\end{array}
\]

as are the sums of the elements in each row, obtained either by:

- Transposing

\[\text{sum}(M') = \]
\[
\begin{array}{ccc}
15 & 15 & 15 \\
\end{array}
\]

- Using the \text{dim} argument

\[\text{sum}(M,1) = \]
\[
\begin{array}{c}
15 \\
15 \\
15 \\
\end{array}
\]

transposing:

**Nondouble Data Type Support**

This section describes the support of \text{sum} for data types other than \text{double}.

**Data Type single**

You can apply \text{sum} to an array of type \text{single} and MATLAB 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 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**

```matlab
 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.

   ```matlab
   load count.dat
   ```

2. Create a timeseries object with 24 time values.

   ```matlab
   count_ts = timeseries(count,1:24,'Name','CountPerSecond')
   ```

3. Calculate the sum of each data column for this timeseries object.

   ```matlab
   sum(count_ts)
sum (timeseries)

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 hierarchy that determines the order in which MATLAB calls object methods.

superiorto('class1', 'class2', ...) invoked within a class constructor method (say myclass.m) indicates that myclass's method should be invoked if a function is called with an object of class myclass and one or more objects of class class1, class2, and so on.

Remarks
Suppose A is of class 'class_a', B is of class 'class_b' and C is of class 'class_c'. Also suppose the constructor class_c.m contains the statement superiorto('class_a'). Then e = fun(a,c) or 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 leftmost 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
support opens the MathWorks Technical Support Web page, http://www.mathworks.com/support, in the MATLAB Web browser. 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',...)

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] = 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.

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.

**Algorithm**

Abstractly, a parametric surface is parameterized by two independent variables, i and j, which vary continuously over a rectangle; for example, 1 ≤ i ≤ m and 1 ≤ j ≤ 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.

```
    i-1,j
   |    
i,j-1 - i,j - i,j+1
```
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 \textit{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 \textit{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 \textit{faceted} (the default) or \textit{flat}, \(C(i,j)\) specifies the constant color in the surface patch:

\[
\begin{array}{c|c}
(i,j) & (i,j+1) \\
\hline
C(i,j) & \\
(i+1,j) & (i+1,j+1)
\end{array}
\]

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 \texttt{surf} and \texttt{surfc} functions specify the viewpoint using \texttt{view(3)}.

The range of \(X\), \(Y\), and \(Z\) or the current setting of the axes \texttt{XLimMode}, \texttt{YLimMode}, and \texttt{ZLimMode} properties (also set by the \texttt{axis} function) determines the axis labels.

The range of \(C\) or the current setting of the axes \texttt{CLim} and \texttt{CLimMode} properties (also set by the \texttt{caxis} function) determines the color scaling. The scaled color values are used as indices into the current colormap.
**surf, surfc**

**Examples**

Display a surfaceplot 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

See Also
axis, caxis, colormap, contour, delaunay, imagesc, mesh, pcolor, shading, trisurf, view
Properties for surfaceplot graphics objects
“Creating Surfaces and Meshes” on page 1-96 for related functions
Representing a Matrix as a Surface for more examples
Coloring Mesh and Surface Plots for information about how to control the coloring of surfaces
surf2patch

**Purpose**
Convert surface data to patch data

**Syntax**

``` matlab
fvc = surf2patch(Z)
fvc = surf2patch(Z,C)
fvc = surf2patch(X,Y,Z)
fvc = surf2patch(X,Y,Z,C)
fvc = surf2patch(...,'triangles')
[f,v,c] = surf2patch(…)
```

**Description**

`fvc = surf2patch(h)`

converts the geometry and color data from the `surface` object identified by the handle `h` into patch format and returns the face, vertex, and color data in the struct `fvc`. You can pass this struct directly to the `patch` command.

`fvc = surf2patch(Z)` calculates the patch data from the surface’s `ZData` matrix `Z`.

`fvc = surf2patch(Z,C)` calculates the patch data from the surface’s `ZData` and `CData` matrices `Z` and `C`.

`fvc = surf2patch(X,Y,Z)` calculates the patch data from the surface’s `XData`, `YData`, and `ZData` matrices `X`, `Y`, and `Z`.

`fvc = surf2patch(X,Y,Z,C)` calculates the patch data from the surface’s `XData`, `YData`, `ZData`, and `CData` matrices `X`, `Y`, `Z`, and `C`.

`fvc = surf2patch(...,'triangles')` creates triangular faces instead of the quadrilaterals that compose surfaces.

`[f,v,c] = 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 `sphere` command to generate the `XData`, `YData`, and `ZData` of a surface, which is then converted to a patch. Note that the `ZData` (z) is passed to `surf2patch` as both the third and fourth arguments — the third argument is the `ZData` and the fourth argument is taken as the `CData`. This is because the `patch` command does not
automatically use the z-coordinate data for the color data, as does the surface command.

Also, because patch is a low-level command, you must set the view to 3-D and shading to faceted to produce the same results produced by the surf command.

```matlab
[x y z] = sphere;
pitch(surf2patch(x,y,z,z));
shading faceted; view(3)
```

In the second example surf2patch calculates face, vertex, and color data from a surface whose handle has been passed as an argument.

```matlab
s = surf(peaks);
pause
patch(surf2patch(s));
delete(s)
shading faceted; view(3)
```

See Also

patch, reducepatch, shrinkfaces, surface, surf

“Volume Visualization” on page 1-101 for related functions
**Purpose**  
Create surface object

**Syntax**

- `surface(Z)`
- `surface(Z,C)`
- `surface(X,Y,Z)`
- `surface(X,Y,Z,C)`
- `surface(x,y,Z)`
- `surface(...'PropertyName',PropertyValue,...)`
- `h = surface(...)`

**Description**

`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.

- `surface(Z)` plots the surface specified by the matrix Z. Here, Z is a single-valued function, defined over a geometrically rectangular grid.

- `surface(Z,C)` plots the surface specified by Z and colors it according to the data in C (see "Examples").

- `surface(X,Y,Z)` uses C = Z, so color is proportional to surface height above the x-y plane.

- `surface(X,Y,Z,C)` plots the parametric surface specified by X, Y, and Z, with color specified by C.

- `surface(x,y,Z)`, `surface(x,y,Z,C)` replaces the first two matrix arguments with vectors and must have `length(x) = n` and `length(y) = m` where `[m,n] = 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 `surf` function.

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

- `h = surface(...)` returns a handle to the created surface object.
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,

    surface('XData',X,'YData',Y,'ZData',Z,'CData',C)

is equivalent to

    surface(X,Y,Z,C)

When you specify only a single matrix input argument,

    surface(Z)

MATLAB assigns the data properties as if you specified

    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

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`. 
Setting Default Properties

You can set default surface properties on the axes, figure, and root 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

“Creating Surfaces and Meshes” on page 1-96 and “Object Creation Functions” on page 1-93 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

*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 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}
**Surface Properties**

*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.

`BackFaceLighting`  
**unlit | lit | reverselit**
Surface Properties

*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
Surface Properties

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**

functional 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(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')
    end
```
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 surface 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{CData}

\begin{itemize}
\item \textbf{matrix (of type double)}
\end{itemize}

\textit{Vertex colors.} A matrix containing values that specify the color
at every point in \texttt{ZData}.

\textbf{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 \texttt{caxis})
or interpreted directly as indices into the colormap, depending on
the setting of the \texttt{CDataMapping} property.

\textbf{CData as True Color}

True color defines an RGB value for each vertex. If the coordinate
data (\texttt{XData}, for example) are contained in \( m \)-by-\( n \) matrices, then
\texttt{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.
Surface Properties

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.

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
functional 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`.

```matlab
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
functional 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 `cla` 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 $\geq 0$ and $\leq 1$

*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.

**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.

- **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
{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.
Surface Properties

- **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>
</tbody>
</table>
Surface Properties

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
### Surface Properties

#### Marker Specifier Description

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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** makes the interior of the marker transparent, allowing the background to show through.
- **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** 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).
Surface Properties

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.
Surface Properties

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

*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 >= 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
Surface Properties

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
**Surface Properties**

_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

_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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaData</td>
<td>m-by-n matrix of <code>double</code> or <code>uint8</code></td>
</tr>
</tbody>
</table>
|                        | *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 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` |

2-3118
**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

\[ \text{scalar} \geq 0 \text{ and } \leq 1 \]

**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.

BackFaceLighting

unlit | lit | reverselit
**Surfaceplot Properties**

*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.

### 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 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.

**CDataMode**

{auto} | manual

*Use automatic or user-specified color data values.* If you specify CData, MATLAB sets this property to manual and uses the CData values to color the surfaceplot.

If you set CDataMode to auto after having specified CData, MATLAB resets the color data of the surfaceplot to that defined by ZData, overwriting any previous values for CData.

**CDataSource**

string (MATLAB variable)

*Link CData to MATLAB variable.* Set this property to a MATLAB variable that 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 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 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.

**DiffuseStrength**

scalar $\geq 0$ and $\leq 1$

*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

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.
Surfaceplot 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 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

*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.

- **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. 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
Surfaceplot Properties

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

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).
**Surfaceplot Properties**

**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>

2-3134
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.
- 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 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).
**Surfaceplot Properties**

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, 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

*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. 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.
Surfaceplot Properties

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.

```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)

*Class of the graphics object.* The class of the graphics object. For surfaceplot objects, Type is always the string 'surface'.

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.

**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.

**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.
**Surfaceplot Properties**

*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)

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.
**Surfaceplot Properties**

**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.
Surface plot with colormap-based lighting

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.

The `surf1` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surf1(Z)` and `surf1(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.

`surf1(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surf1(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surf1(...,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.

`surf1(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
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.

    [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`

“Creating Surfaces and Meshes” on page 1-96 for functions related to surfaces

“Lighting” on page 1-100 for functions related to lighting
Purpose

Compute and display 3-D surface normals

Syntax

```
surfnorm(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

The direction of the normals is reversed by calling `surfnorm` with transposed arguments:

```
surfnorm(X',Y',Z')```

`surf1` 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.

```matlab
[x,y,z] = cylinder(1:10);
surfnorm(x,y,z)
axis([-12 12 -12 12 -0.1 1])
```

See Also

surf, quiver3

“Colormaps” on page 1-98 for related functions
**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

\[
X =
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
5 & 6 \\
7 & 8
\end{bmatrix}
\]

the statement

\[
[U,S,V] = svd(X)
\]

produces

\[
U =
\begin{bmatrix}
-0.1525 & -0.8226 & -0.3945 & -0.3800
\end{bmatrix}
\]
svd

\[
\begin{bmatrix}
-0.3499 & -0.4214 & 0.2428 & 0.8007 \\
-0.5474 & -0.0201 & 0.6979 & -0.4614 \\
-0.7448 & 0.3812 & -0.5462 & 0.0407 \\
\end{bmatrix}
\]

\[S = \begin{bmatrix}
14.2691 & 0 \\
0 & 0.6268 \\
0 & 0 \\
0 & 0 \\
\end{bmatrix}\]

\[V = \begin{bmatrix}
-0.6414 & 0.7672 \\
-0.7672 & -0.6414 \\
\end{bmatrix}\]

The economy size decomposition generated by

\[[U,S,V] = \text{svd}(X,0)\]

produces

\[U = \begin{bmatrix}
-0.1525 & -0.8226 \\
-0.3499 & -0.4214 \\
-0.5474 & -0.0201 \\
-0.7448 & 0.3812 \\
\end{bmatrix}\]

\[S = \begin{bmatrix}
14.2691 & 0 \\
0 & 0.6268 \\
\end{bmatrix}\]

\[V = \begin{bmatrix}
-0.6414 & 0.7672 \\
-0.7672 & -0.6414 \\
\end{bmatrix}\]

**Algorithm**

svd uses the LAPACK routines listed in the following table to compute the singular value decomposition.
### Diagnostics

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

Solution will not converge.

### References

svds

### 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 = svds(A)` computes the six largest singular values and associated singular vectors of matrix `A`. If `A` is `m`-by-`n`, `svds(A)` manipulates eigenvalues and vectors returned by `eigs(B)`, where `B = [sparse(m,m) A; A' 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 = svds(A,k)` computes the `k` largest singular values and associated singular vectors of matrix `A`.

- `s = svds(A,k,sigma)` computes the `k` singular values closest to the scalar shift `sigma`. For example, `s = svds(A,k,0)` computes the `k` smallest singular values and associated singular vectors.

- `s = svds(A,k,'L')` computes the `k` largest singular values (the default).

- `s = svds(A,k,sigma,options)` sets some parameters (see `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: <code>norm(AV-US,1)&lt;=tol*norm(A,1)</code></td>
<td><code>1e-10</code></td>
</tr>
<tr>
<td>options.maxit</td>
<td>Maximum number of iterations</td>
<td><code>300</code></td>
</tr>
<tr>
<td>options.disp</td>
<td>Number of values displayed each iteration</td>
<td><code>0</code></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 \( \|A*V - U*S\|_1 \leq tol*\|A\|_1 \) 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.

```matlab
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:

\[
\text{svds}(\text{west0479},1) = 3.189517598808622e+05 \\
\max(\text{svd}(\text{full}(\text{west0479}))) = 3.18951759880862e+05 \\
\text{norm}(\text{full}(\text{west0479})) = 3.189517598808623e+05
\]

and estimated:

\[
\text{normest}(\text{west0479}) = 3.189385666549991e+05
\]

**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:

\[
A = \text{uint32}(%2812345678\%29);  \\
B = \text{dec2hex}(\text{swapbytes}(A))  \\
\]

\[ B = 78563412 \]

**Example 2**
Reverse the byte order for each element of a 1-by-4 matrix:

\[
X = \text{uint16}([0 1 128 65535])  \\
X =  \\
\begin{array}{cccc}
0 & 1 & 128 & 65535 \\
\end{array}  \\
\]

\[
Y = \text{swapbytes}(X);  \\
Y =  \\
\begin{array}{cccc}
0 & 256 & 32768 & 65535 \\
\end{array}  \\
\]

Examining the output in hexadecimal notation shows the byte swapping:

\[
\text{format hex}  \\
X, Y  \\
X =  \\
\begin{array}{cccc}
0000 & 0001 & 0080 & ffff \\
\end{array}  \\
\]
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)
ans(:,:,1) =
    b004 9600 8403
    c201 ee02 1a04
    5802 4605 2c01
ans(:,:,2) =
    80bb 7017 a08c
    5046 3075 10a4
    c05d f0d2 e02e
```

See Also

typecast
### Purpose
Switch among several cases, based on expression

### Syntax

```
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).
switch

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
Symmetric approximate minimum degree permutation

Syntax

\[ p = \text{symamd}(S) \]
\[ p = \text{symamd}(S, \text{knobs}) \]
\[ [p, \text{stats}] = \text{symamd}(\ldots) \]

Description

\[ p = \text{symamd}(S) \] 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 \( S \), \text{symamd} constructs a matrix \( M \) such that \( \text{spones}(M' * M) = \text{spones}(S) \), and then computes \( p = \text{colamd}(M) \). The \text{symamd} function may also work well for symmetric indefinite matrices.

\( S \) must be square; only the strictly lower triangular part is referenced.

\[ p = \text{symamd}(S, \text{knobs}) \] where \( \text{knobs} \) is a scalar. If \( S \) is \( n \)-by-\( n \), rows and columns with more than \( \text{knobs} * n \) entries are removed prior to ordering, and ordered last in the output permutation \( p \). If the \( \text{knobs} \) parameter is not present, then \( \text{knobs} = \text{spparms('wh_frac')} \).

\[ [p, \text{stats}] = \text{symamd}(\ldots) \] produces the optional vector \( \text{stats} \) that provides data about the ordering and the validity of the matrix \( S \).

\begin{align*}
\text{stats}(1) & \quad \text{Number of dense or empty rows ignored by symamd} \\
\text{stats}(2) & \quad \text{Number of dense or empty columns ignored by symamd} \\
\text{stats}(3) & \quad \text{Number of garbage collections performed on the internal data structure used by symamd (roughly of size } 8.4 * \text{nnz(tril}(S,-1)) + 9n \text{ integers)} \\
\text{stats}(4) & \quad 0 \text{ if the matrix is valid, or } 1 \text{ if invalid} \\
\text{stats}(5) & \quad \text{Rightmost column index that is unsorted or contains duplicate entries, or } 0 \text{ if no such column exists} \\
\text{stats}(6) & \quad \text{Last seen duplicate or out-of-order row index in the column index given by stats}(5), \text{ or } 0 \text{ if no such row index exists} \\
\text{stats}(7) & \quad \text{Number of duplicate and out-of-order row indices}
\end{align*}
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.

**Note** symamd tends to be faster than symmmd and tends to return a better 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

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
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=chol(A'*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=chol(A'*A)
(without forming it explicitly).

count = symbfact(A,'row') returns row counts of R=chol(A*A').

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 chol(A) for the symmetric case, chol(A'<em>A) for the 'col' case, or chol(A</em>A') 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**

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

**Description**

\(x = \text{symmlq}(A,b)\) 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 \(\text{afun}\) such that \(\text{afun}(x)\) returns \(A*x\). See “Function Handles” in the MATLAB Programming documentation for more information.

“Parameterizing Functions Called by Function 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 \(\text{symmlq}\) converges, a message to that effect is displayed. If \(\text{symmlq}\) 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.

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

\(\text{symmlq}(A,b,tol,maxit)\) specifies the maximum number of iterations. If \(maxit\) is [], then \(\text{symmlq}\) 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 $M\times x$.

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,flag] = symmlq(A,b,...) 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,flag,relres] = symmlq(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = symmlq(A,b,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = symmlq(A,b,...) also returns a vector of estimates of the symmlq residual norms at each iteration, including 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**

```matlab
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);
```
symmlq

```
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
```

When you enter
```
x1=run_symmlq;
```
MATLAB 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 symmetric minimum degree ordering

Syntax
\[ p = \text{symmmd}(S) \]

Note
\text{symmmd} is obsolete and will be removed from a future version of MATLAB. Use \text{symamd} instead.

Description
\( p = \text{symmmd}(S) \) returns a symmetric minimum degree ordering of \( S \). For a symmetric positive definite matrix \( S \), this is a permutation \( p \) such that \( S(p,p) \) tends to have a sparser Cholesky factor than \( S \). Sometimes \text{symmmd} works well for symmetric indefinite matrices too.

Algorithm
The symmetric minimum degree algorithm is based on the column minimum degree algorithm. In fact, \text{symmmd}(A) just creates a nonzero structure \( K \) such that \( K' * K \) has the same nonzero structure as \( A \) and then calls the column minimum degree code for \( K \).

See Also
\text{colamd}, \text{colmmd}, \text{colperm}, \text{symamd}, \text{symrcm}

References
Purpose  
Sparse reverse Cuthill-McKee ordering

Syntax
\[ r = \text{symrcm}(S) \]

Description  
\( r = \text{symrcm}(S) \) 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 \( \text{eig}(S(r,r)) \) probably takes less time to compute than \( \text{eig}(S) \).

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
\[ B = \text{bucky}; \]
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

\[ \text{subplot}(1,2,1), \text{spy}(B), \text{title('B')} \]

The reverse Cuthill-McKee ordering is obtained with
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)')

This example is continued in the reference pages for symamd.

The bandwidth can also be computed with

[i,j] = find(B);
bw = max(i-j) + 1;

The bandwidths of B and R are 35 and 12, respectively.

See Also
colamd, colperm, 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 =

'beta1'
'x'
```

**See Also**
findstr
**Purpose**
Synchronize and resample two timeseries objects using common time vector.

**Syntax**
\[ [ts1\ ts2] = synchronize(ts1,ts2,'SynchronizeMethod') \]

**Description**
\[ [ts1\ ts2] = synchronize(ts1,ts2,'SynchronizeMethod') \] 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:
  \[ [ts1\ ts2] = synchronize(ts1,ts2,'Uniform','Interval',value) \]

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 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 2-3177.

**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:
Arguments are passed to the function by value. See the examples below, under “Argument Passing” on page 2-3177.

**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':

```matlab
A = pi;
disp(A)
A
```

while function syntax passes the value assigned to `A`:

```matlab
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':

```matlab
str1 = 'one'; str2 = 'one';
strcmp str1 str2
ans =
0 (unequal)
```

while function syntax compares the values assigned to the variables, 'one' and 'one':

```matlab
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
### Purpose
Execute operating system command and return result.

### Syntax
```matlab
system('command')
[status, result] = system('command')
```

### 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 MATLAB. 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.

```matlab
[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 Windows 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:

```plaintext
??? 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.

```matlab
[status currdir] = system('cd')
status =
    0
currdir =
```

2-3179
system

D:\work\matlab\test

See Also

! (bang), computer, dos, perl, unix, winopen

“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 \(-\pi/2 < x < \pi/2\).  

\[
\begin{align*}
x &= (-\pi/2)+0.01:0.01:(\pi/2)-0.01; \\
\text{plot}(x,\tan(x)), \text{ grid on}
\end{align*}
\]

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

\[ \tan(x) = \frac{\sin(x)}{\cos(x)} \]
\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**

\[ Y = \text{tand}(X) \]

**Description**

\[ Y = \text{tand}(X) \] is the tangent of the elements of \( X \), expressed in degrees. For odd integers \( n \), \( \text{tand}(n*90) \) is infinite, whereas \( \tan(n*pi/2) \) is large but finite, reflecting the accuracy of the floating point value of \( \pi \).

**See Also**

\( \tan, \tanh, \text{atan}, \text{atan2}, \text{atand}, \text{atanh} \)
tanh

**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 \(-5 \leq x \leq 5\).

\[
x = -5:0.01:5;
plot(x,\tanh(x)), grid on
\]

**Definition**
The hyperbolic tangent can be defined as

\[
\tanh(z) = \frac{\sinh(z)}{\cosh(z)}
\]
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 with the name 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. The .tar extension is appended to tarfilename if omitted. 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 pathnames. 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 filename or partial pathname.

- tar(tarfilename,files,rootdir) allows the path for files to be specified relative to rootdir rather than the current directory.

- entrynames = tar(...) returns a string cell array of the relative path entry names contained in tarfilename.

**Example**
Tar all files in the current directory to the file backup.tgz:
```
tar('backup.tgz','.');
```

**See Also**
gzip, gunzip, untar, unzip, zip
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 n tetrahedrons. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

h = 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'.

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.

```matlab
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

tetramesh(Tes,X);camorbit(20,0)
```
See Also
delaunay, 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 1-86 for related functions
text

Purpose
Create text object in current axes

Syntax

\[
\text{text}(x,y,'string')
\]
\[
\text{text}(x,y,z,'string')
\]
\[
\text{text}(x,y,z,'string','PropertyName',PropertyValue....)
\]
\[
\text{text('PropertyName',PropertyValue....)}
\]
\[
h = \text{text}(...)
\]

Description
text is the low-level function for creating text graphics objects. Use text to place character strings at specified locations.

\[\text{text}(x,y,'string')\] adds the string in quotes to the location specified by the point \((x,y)\).

\[\text{text}(x,y,z,'string')\] adds the string in 3-D coordinates.

\[\text{text}(x,y,z,'string','PropertyName',PropertyValue....)\] 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 section at the end of this page for a list of text properties.

\[\text{text('PropertyName',PropertyValue....)}\] omits the coordinates entirely and specifies all properties using property name/property value pairs.

\[h = \text{text}(...)\] 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 “Example”). You can use other units to position the text by set 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
- A string vector using vertical slash characters ('|') as separators.

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,

\[
\text{text}(x,y,z,\text{string'})
\]

is equivalent to

\[
\text{text('Position',[x,y,z],'String',\text{string'})}
\]
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

\texttt{text(x,y,'\textstyle{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 levels:

```matlab
set(0,'DefaulttextProperty',PropertyValue...)  
set(gcf,'DefaulttextProperty',PropertyValue...)  
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`, `title`, `xlabel`, `ylabel`, `zlabel`, `strings`  
“Object Creation Functions” on page 1-93 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.

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.

```
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.

**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 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
functional 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
```

Suppose `h` is the handle of a text object and that the `button_down` function is on your MATLAB path. The following statement assigns the function above to the `ButtonDownFcn`:

```matlab
set(h, 'ButtonDownFcn', @button_down)
```
See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

Children
matrix (read only)

The empty matrix; text objects have no children.

Clipping
on | {off}

*Clipping mode.* When Clipping is on, MATLAB does not display any portion of the text that is outside the axes.

Color
ColorSpec

*Text color.* A three-element RGB vector or one of the predefined names, specifying the text color. The default value for Color is white. See ColorSpec for more information on specifying color.

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

*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 text function that creates a new text object. For example, the statement

```
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.

```
function text_create(src,evnt)
```
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**

Functional 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 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.

**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),...'
\textarrow{\text{sin(t) = .707}},...
'EdgeColor','red');
```
For additional features, see the following 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

\[ [\text{left}, \text{bottom}, \text{width}, \text{height}] \]

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:

```
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
Text Properties

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 `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.
- 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`
**Text Properties**

*Selectable by mouse click.* HitTest determines if the text 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 text. If HitTest is set to 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 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 T_{\text{E}X} instructions.* This property controls whether MATLAB interprets certain characters in the String property as T_{\text{E}X} instructions (default) or displays all characters literally. The options are:

- latex — Supports the full L_{\text{A}T\text{E}X} markup language.
- tex — Supports a subset of plain T_{\text{E}X} markup language. See the String property for a list of supported T_{\text{E}X} instructions.
- none — Displays literal characters.

**Latex Interpreter**

To enable the LaT_{\text{E}X} interpreter for text objects, set the Interpreter property to latex. For example, the following statement displays an equation in a figure at the point [.5 .5], and enlarges the font to 16 points.

```latex
\text{'Interpreter','latex',... \\
'String','$$\int_0^x \int_y dF(u,v)$$',... \\
'Position',[.5 .5],... \\
'FontSize',16)}
```
Information About Using TEX

The following references may be useful to people who are not familiar with $T_{\text{E}}X$.

- The $T_{\text{E}}X$ 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><strong>Symbol</strong></th>
<th><strong>Line Style</strong></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',':' );
```
Text Properties

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);
```

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.

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>\leftrightharpoon</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>i</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>Υ</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>
<tr>
<td>\nu</td>
<td>ν</td>
<td>\Phi</td>
<td>φ</td>
<td>\pm</td>
<td>±</td>
</tr>
<tr>
<td>\xi</td>
<td>ξ</td>
<td>\Psi</td>
<td>Ψ</td>
<td>\geq</td>
<td>≥</td>
</tr>
<tr>
<td>\pi</td>
<td>π</td>
<td>\Omega</td>
<td>Ω</td>
<td>\propto</td>
<td>∝</td>
</tr>
<tr>
<td>\rho</td>
<td>ρ</td>
<td>\forall</td>
<td>∀</td>
<td>\partial</td>
<td>∂</td>
</tr>
<tr>
<td>\sigma</td>
<td>σ</td>
<td>\exists</td>
<td>∃</td>
<td>\bullet</td>
<td>•</td>
</tr>
<tr>
<td>\varsigma</td>
<td>ς</td>
<td>\ni</td>
<td>ι</td>
<td>\div</td>
<td>÷</td>
</tr>
<tr>
<td>\tau</td>
<td>τ</td>
<td>\cong</td>
<td>≈</td>
<td>\neq</td>
<td>≠</td>
</tr>
<tr>
<td>\equiv</td>
<td>≡</td>
<td>\approx</td>
<td>~</td>
<td>\aleph</td>
<td>ℵ</td>
</tr>
<tr>
<td>\Im</td>
<td>ℑ</td>
<td>\Re</td>
<td>ℜ</td>
<td>\wp</td>
<td>ϝ</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 \texttt{\color} modifier to change the color of characters following it from the previous color (which is black by default). Syntax is:

- \texttt{\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 \texttt{\color} modifier.

- \texttt{\color\{rgb\}\{r g b\}} to specify an RGB triplet with values between 0 and 1 as a cell array

For example,

```latex
\text{text(.1,.5,['\fontsize{16}{16}\color{black}black \color{magenta}magenta \ldots \color{rgb}{0 .5 .5}teal \color{red}red} black again')}\)
```

2-3223
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: \\\n\\, \_, \^.

See the “Examples” on page 2-3196 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 = $\frac{1}{72}$ 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 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.
### Purpose
Read data from text file; write to multiple outputs

### Note
The `textscan` function is intended as a replacement for both `textread` and `strread`.

### 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.

textrread 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</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 <code>format</code> 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 <code>%f</code> format supports <code>%w.pf</code>, 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

 [...] = 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></td>
<td>' ' \b \n \r \t</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Backspace</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Newline</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Carriage return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Horizontal tab</td>
</tr>
<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>
</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', '')
ans =
    ' 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 =
    12.34000000000000
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.

\[
\text{[names, types, y, answer] = textread('mydata.dat', ...}
\text{'%9c %5s %*f %2d %3s', 1)}
\]

returns

\[
\begin{align*}
\text{names} &= \\
\text{Sally} & \\
\text{types} &= \\
\text{'Level1'} & \\
\text{y} &= \\
\text{45} & \\
\text{answer} &= \\
\text{'Yes'} &
\end{align*}
\]

%*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.

\[
\text{[names, typenum, x, y, answer] = textread('mydata.dat', ...}
\text{'%s Type%d %f %d %s', 1)}
\]
returns

    names = 'Sally'
    typenum = 1
    x = 12.34000000000000
    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:

    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.

    file = textread('fft.m', '%s', 'delimiter', '\n', ...
                    'whitespace', '');

See Also
textscan, dlmread, csvread, strread, 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, you should 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. MATLAB parses the data into fields and converts it according to the conversion specifiers in format. The format input is a string enclosed in single quotes. These conversion specifiers determine the type of each cell in the output cell array. The number of specifiers determines the number of cells in the cell array.

C = textscan(fid, 'format', N) reads data from the file, reusing the format conversion specifier N times, where N is a positive integer. You can resume reading 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. The section “User Configurable Options” on page 2-3243 lists all valid parameter strings, value descriptions, and defaults.

C = textscan(fid, 'format', N, param, value, ...) reads data from the file, reusing the format conversion specifier N times, and using
nondefault parameter settings specified by pairs of `param` and `value` arguments.

\[ C = \text{textscan}(\text{str}, ...) \] reads data from string `str` in exactly the same way as it does when reading from a file. You can use the `format`, `N`, and parameter/value arguments described above with this syntax. Unlike when reading from a file, if you call `textscan` more than once on the same string, it does not resume reading where the last call left off but instead reads from the beginning of the string each time.

\[ [C, \text{position}] = \text{textscan}(...) \] returns the location of the file or string position as the second output argument. For a file, this is exactly equivalent to calling `fseek(fid)` after making the call to `textscan`. For a string, it indicates how many characters were read.

**The Difference Between the textscan and textread Functions**

The `textscan` function differs from `textread` in the following ways:

- The `textscan` function offers better performance than `textread`, making it a better choice when reading large files.
- With `textscan`, you can start reading at any point in the file. Once the file is open, (`textscan` requires that you open the file first), you can `fseek` to any position in the file and begin the scan at that point. The `textread` function requires that you start reading from the beginning of the file.
- Subsequent `textscan` operations start reading the file at the point where the last scan left off. The `textread` function always begins at the start of the file, regardless of any prior `textread` operations.
- `textscan` returns a single cell array regardless of how many fields you read. With `textscan`, you don’t need to match the number of output arguments to the number of fields being read as you would with `textread`.
- `textscan` offers more choices in how the data being read is converted.
- `textscan` offers more user-configurable options.
Field Delimiters

The `textscan` function sees a text file as a collection of blocks. Each block consists of a number of internally consistent fields. Each field consists of a group of characters delimited by a field delimiter character. Fields can span a number of rows. Each row is delimited by an end-of-line (EOL) character sequence.

The default field delimiter is the white-space character, (i.e., any character that returns true from a call to the `isspace` function). You can set the delimiter to a different character by specifying a 'delimiter' parameter in the `textscan` command (see “User Configurable Options” on page 2-3243). If a nondefault delimiter is specified, repeated delimiter characters are treated as separate delimiters. When using the default delimiter, repeated white-space characters are treated as a single delimiter.

The default end-of-line character sequence depends on which operating system you are using. You can change the end-of-line setting to a different character sequence by specifying an 'endofline' parameter in the `textscan` command (see “User Configurable Options” on page 2-3243).

Conversion Specifiers

This table shows the conversion type specifiers supported by `textscan`.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%n</td>
<td>Read a number and convert to <code>double</code>.</td>
</tr>
<tr>
<td>%d</td>
<td>Read a number and convert to <code>int32</code>.</td>
</tr>
<tr>
<td>%d8</td>
<td>Read a number and convert to <code>int8</code>.</td>
</tr>
<tr>
<td>%d16</td>
<td>Read a number and convert to <code>int16</code>.</td>
</tr>
<tr>
<td>%d32</td>
<td>Read a number and convert to <code>int32</code>.</td>
</tr>
<tr>
<td>%d64</td>
<td>Read a number and convert to <code>int64</code>.</td>
</tr>
<tr>
<td>%u</td>
<td>Read a number and convert to <code>uint32</code>.</td>
</tr>
<tr>
<td>Specifier</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>%u8</td>
<td>Read a number and convert to uint8.</td>
</tr>
<tr>
<td>%u16</td>
<td>Read a number and convert to uint16.</td>
</tr>
<tr>
<td>%u32</td>
<td>Read a number and convert to uint32.</td>
</tr>
<tr>
<td>%u64</td>
<td>Read a number and convert to uint64.</td>
</tr>
<tr>
<td>%f</td>
<td>Read a number and convert to double.</td>
</tr>
<tr>
<td>%f32</td>
<td>Read a number and convert to single.</td>
</tr>
<tr>
<td>%f64</td>
<td>Read a number and convert to double.</td>
</tr>
<tr>
<td>%s</td>
<td>Read a string.</td>
</tr>
<tr>
<td>%q</td>
<td>Read a (possibly double-quoted) string.</td>
</tr>
<tr>
<td>%c</td>
<td>Read one character, including white space.</td>
</tr>
<tr>
<td>[%... ]</td>
<td>Read characters that match characters between the brackets. Stop reading at the first nonmatching character. Use %[ ]... to include } in the set.</td>
</tr>
<tr>
<td>[%^... ]</td>
<td>Read characters that do not match characters between the brackets. Stop reading at the first matching character. Use %[^]... to exclude } from the set.</td>
</tr>
<tr>
<td>%*n...</td>
<td>Ignore n characters of the field, where n is an integer less than or equal to the number of characters in the field (e.g., %*4s).</td>
</tr>
</tbody>
</table>

**Specifying Field Length**

To read a certain number of characters or digits from a field, specify that number directly following the percent sign. For example, if the file you are reading contains the string

    'Blackbird singing in the dead of night'

then the following command returns only five characters of the first field:
C = textscan(fid, '%5s', 1);
C{::}
ans =
'Black'

If you continue reading from the file, textscan resumes the operation at the point in the string where you left off. It applies the next format specifier to that portion of the field. For example, execute this command on the same file:

C = textscan(fid, '%s %s', 1);

Note Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

textscan reads starting from where it left off and continues to the next whitespace, returning 'bird'. The second %s reads the word 'singing'.
The results are

C{::}
ans =
'bird'
ans =
'singing'

Skipping Fields
To skip any field, put an asterisk directly after the percent sign. MATLAB does not create an output cell for any fields that are skipped.

Refer to the example from the last section, where the file you are reading contains the string

'Blackbird singing in the dead of night'
Seek to the beginning of the file and reread the line, this time skipping the second, fifth, and sixth fields:

\[
\text{fseek(fid, 0, -1);}
\]
\[
C = \text{textscan(fid, 's s s s s s', 1)};
\]

C is a cell array of cell arrays, each containing a string. Piece together the string and display it:

\[
\text{str} = '';\]
\[
\text{for} \ k = 1:\text{length(C)}\]
\[
\text{\quad str} = [\text{str char(C{\{k\}}} ' '];\]
\[
\text{\quad if} \ k == 4, \ \text{disp(str)}, \ \text{end}\]
\[
\text{end}\]

Blackbird in the night

**Skipping Literal Strings**

In addition to skipping entire fields, you can have `textscan` skip leading literal characters in a string. Reading a file containing the following data,

<table>
<thead>
<tr>
<th>Name</th>
<th>Level</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sally</td>
<td>Level1</td>
<td>12.34</td>
</tr>
<tr>
<td>Joe</td>
<td>Level2</td>
<td>23.54</td>
</tr>
<tr>
<td>Bill</td>
<td>Level3</td>
<td>34.90</td>
</tr>
</tbody>
</table>

this command removes the substring 'Level' from the output and converts the level number to a `uint8`:

\[
C = \text{textscan(fid, 's Level\%u8 \%f');}
\]

This returns a cell array C with the second cell containing only the unsigned integers:

\[
C{\{1\}} = \{'Sally'; 'Joe'; 'Bill'} \quad \text{class cell}
\]
\[
C{\{2\}} = [1; 2; 3] \quad \text{class uint8}
\]
\[
C{\{3\}} = [12.34; 23.54; 34.90] \quad \text{class double}
\]
Specifying Numeric Field Length and Decimal Digits

With numeric fields, you can specify the number of digits to read in the same manner described for strings in the section “Specifying Field Length” on page 2-3237. The next example uses a file containing the line

'405.36801  551.94387  298.00752  141.90663'

This command returns the starting 7 digits of each number in the line. Note that the decimal point counts as a digit.

```matlab
C = textscan(fid, '%7f32 %*n');
C{:} =
    [405.368; 551.943; 298.007; 141.906]
```

You can also control the number of digits that are read to the right of the decimal point for any numeric field of type %f, %f32, or %f64. The format specifier in this command uses a %9.1 prefix to cause textscan to read the first 9 digits of each number, but only include 1 digit of the decimal value in the number it returns:

```matlab
C = textscan(fid, '%9.1f32 %*n');
C{:} =
    [405.3; 551.9; 298.0; 141.9]
```

Conversion of Numeric Fields

This table shows how textscan interprets the numeric field specifiers.

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>%n, %d, %u, %f, and variants thereof</td>
<td>Read to the first delimiter.</td>
</tr>
<tr>
<td></td>
<td>Example: %n reads '473.238 ' as 473.238.</td>
</tr>
<tr>
<td>Format Specifier</td>
<td>Action Taken</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>%Nn, %Nd, %Nu, %Nf, and variants thereof</td>
<td>Read N digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Example: %5f32 reads '473.238 ' as 473.2.</td>
</tr>
<tr>
<td>Specifiers that start with %N.Df</td>
<td>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. Example: %7.2f reads '473.238 ' as 473.23.</td>
</tr>
</tbody>
</table>

Conversion specifiers %n, %d, %u, %f, or any variant thereof (e.g., %d16) return a K-by-1 MATLAB numeric vector of the type indicated by the conversion specifier, where K is the number of times that specifier was found in the file. textscan converts the numeric fields from the field content to the output type according to the conversion specifier and MATLAB rules regarding overflow and truncation. NaN, Inf, and -Inf are converted according to applicable MATLAB rules.

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

<table>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&lt;real&gt;--&lt;imag&gt;i</td>
<td>j</td>
</tr>
<tr>
<td>--&lt;imag&gt;i</td>
<td>j</td>
</tr>
</tbody>
</table>

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

**Conversion of Strings**

This table shows how textscan interprets the string field specifiers.
### Format Specifier

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Action Taken</th>
</tr>
</thead>
</table>
| %s or %q         | Read to the first delimiter.  
Example: "%s reads 'summer' as 'summer'". |
| %Ns or %Nq       | Read N characters, or to the first delimiter, whichever comes first.  
Example: "%3s reads 'summer' as 'sum'". |
| %[abc]           | Read those characters that match any character specified within the brackets, stopping just before the first character that does not match.  
Example: "%[mus] reads 'summer' as 'summ'". |
| %N[abc]          | Read as many as N characters that match any character specified within the brackets, stopping just before the first character that does not match.  
Example: "%2[mus] reads 'summer' as 'su'". |
| %[^abc]          | Read those characters that do not match any character specified within the brackets, stopping just before the first character that does match.  
Example: "%[^xrg] reads 'summer' as 'summe'". |
| %N[^abc]         | Read as many as N characters that do not match any character specified within the brackets, stopping just before the first character that does match.  
Example: "%2[^xrg] reads 'summer' as 'su'". |

Conversion specifiers %s, %q, [% ...], and %[^ ...] return a K-by-1 MATLAB cell vector of strings, where K is the number of times that specifier was found in the file. If you set the delimiter parameter to a non-white-space character, or set the whitespace parameter to '', textscan returns all characters in the string field, including white-space. Otherwise each string terminates at the beginning of white-space.
Conversion of Characters

This table shows how textscan interprets the character field specifiers.

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Action Taken</th>
</tr>
</thead>
</table>
| %c               | Read one character.  
Example: %c reads 'Let's go!' as 'L'. |
| %Nc              | Read N characters, including delimiter characters.  
Example: %9c reads 'Let's go!' as 'Let's go!'. |

Conversion specifier %Nc returns a K-by-N MATLAB character array, where K is the number of times that specifier was found in the file. textscan returns all characters, including white-space, but excluding the delimiter.

Conversion of Empty Fields

An empty field in the text file is defined by two adjacent delimiters indicating an empty set of characters, or, in all cases except %c, white-space. The empty field is returned as NaN by default, but is user definable. In addition, you may specify custom strings to be used as empty values, in numeric fields only. textscan does not examine nonnumeric fields for custom empty values. See “User Configurable Options” on page 2-3243.

Note MATLAB represents integer NaN as zero. If textscan reads an empty field that is assigned an integer format specifier (one that starts with %d or %u), it returns the empty value as zero rather than as NaN. (See the value returned in C{5} in Example 6 — Using a Nondefault Empty Value.

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>Parameter</th>
<th>Value</th>
<th>Default</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, MATLAB concatenates consecutive cells of the output that have the same data type into a single array.</td>
<td>0 (false)</td>
</tr>
<tr>
<td>CommentStyle</td>
<td>Symbol(s) designating text to be ignored (see “Values for commentStyle” on page 2-3245, below)</td>
<td>None</td>
</tr>
<tr>
<td>Delimiter</td>
<td>Delimiter characters</td>
<td>Whitespace</td>
</tr>
<tr>
<td>EmptyValue</td>
<td>Empty cell value in delimited files</td>
<td>NaN</td>
</tr>
<tr>
<td>endOfLine</td>
<td>End-of-line character</td>
<td>Determined from the file</td>
</tr>
<tr>
<td>expChars</td>
<td>Exponent characters</td>
<td>'eEdD'</td>
</tr>
<tr>
<td>HeaderLines</td>
<td>Number of lines at beginning of file to skip</td>
<td>0</td>
</tr>
<tr>
<td>MultipleDelimsAsOne</td>
<td>If set to 1, textread treats consecutive delimiters as a single delimiter. If set to 0, textread treats them as separate delimiters. Only valid if the delimiter option is specified.</td>
<td>0</td>
</tr>
<tr>
<td>ReturnOnError</td>
<td>Behavior on failing to read or convert (1=true, or 0)</td>
<td>1</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>TreatAsEmpty</td>
<td>String(s) to be treated as an empty value. A single string or cell array of strings can be used.</td>
<td>None</td>
</tr>
<tr>
<td>Whitespace</td>
<td>White-space characters</td>
<td>' \b\t'</td>
</tr>
</tbody>
</table>

**White-Space Characters**

Leading white-space characters are not included in the processing of any of the data fields. When processing numeric data, trailing whitespace is also assumed to have no significance.

**Values for commentStyle**

Possible values for the `commentStyle` parameter are

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single string, S</td>
<td>Ignore any characters that follow string S and are on the same line.</td>
<td>'%', '//'</td>
</tr>
<tr>
<td>Cell array of two strings, C</td>
<td>Ignore any characters that lie between the opening and closing strings in C.</td>
<td>{'/<em>', '</em>/'}, {'/%', '%/'}</td>
</tr>
</tbody>
</table>

**Resuming a Text Scan**

If `textscan` fails to convert a data field, it stops reading and returns all fields read before the failure. When reading from a file, you can resume reading from the same file by calling `textscan` again using the same file identifier, `fid`. When reading from a string, the two-output argument syntax enables you to resume reading from the string at the point where the last read terminated. The following command is an example of how you can do this:
textscan

textscan(str(position+1:end), ...)

Remarks

For information on how to use textscan to import large data sets, see “Reading Files with Large Data Sets” in the MATLAB Programming documentation.

Examples

Example 1 — Reading Different Types of Data

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

Read each column into a variable:

```matlab
fid = fopen('scan1.dat');
C = textscan(fid, '%s %s %f32 %d8 %u %f %f %s');
fclose(fid);
```

Note

Spaces between the conversion specifiers are shown only to make the example easier to read. They are not required.

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 2 — Reading All But One Field

Read the file as a fixed-format file, skipping the third field:

```matlab
fid = fopen('scan1.dat');
C = textscan(fid, '%7c %6s %*f %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 char
- `C{2} = {'Level1'; 'Level2'; 'Level3'}` class cell
- `C{3} = [45; 60; 12]` class int8
- `C{4} = [4294967295; 4294967295; 200000]` class uint32
- `C{5} = [Inf; -Inf; 10]` class double
- `C{6} = [NaN; 0.001; 100]` class double
- `C{7} = {'Yes'; 'No'; 'No'}` class cell

Example 3 — Reading Only the First Field

Read the first column into a cell array, skipping the rest of the line:

```matlab
fid = fopen('scan1.dat');
names = textscan(fid, '%s%*[^
\]');
fclose(fid);
```

textscan returns a 1-by-1 cell array `names`:

- `size(names)`
  - `ans =
  
    1     1`

The one cell contains

- `names{1} = {'Sally'; 'Joe'; 'Bill'}` class cell

Example 4 — Removing a Literal String in the Output

The second format specifier in this example, `%sLevel`, tells textscan to read the second field from a line in the file, but to ignore the initial string 'Level' within that field. All that is left of the field is a numeric
digit. textscan assigns the next specifier, %f, to that digit, converting it to a double.

See C{2} in the results:

```
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 cells

```
C{1} = {'Sally'; 'Joe'; 'Bill'} class cell
C{2} = [1; 2; 3] class uint8
C{3} = [12.34; 23.54; 34.90] 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
```

**Example 5 — Using a Nondefault Delimiter and White-Space**

Read the M-file into a cell array of strings:

```
fid = fopen('fft.m');
file = textscan(fid, '%s', 'delimiter', '
', ... 'whitespace', '');
fclose(fid);
```

textscan returns a 1-by-1 cell array, file, that contains a 37-by-1 cell array:

```
file =
{37x1 cell}
```

Show some of the text from the first three lines of the file:

```
lines = file{1};
lines{1:3, :}
an =
```
%FFT Discrete Fourier transform.
ans =
% FFT(X) is the discrete Fourier transform (DFT) of vector X. For
ans =
% matrices, the FFT operation is applied to each column. For N-D

Example 6 — Using a Nondefault Empty Value

Read files with empty cells, setting the emptyvalue parameter. The file data.csv contains

```
1, 2, 3, 4, , 6
7, 8, 9, , 11, 12
```

Read the file as shown here, using -Inf in empty cells:

```
fid = fopen('data.csv');
C = textscan(fid, '%f%f%f%f%u32%f', 'delimiter', ',', ...
    'emptyValue', -Inf);
fclose(fid);
```

textscan returns a 1-by-6 cell array C with the following cells:

```
C{1} = [1; 7]        class double
C{2} = [2; 8]        class double
C{3} = [3; 9]        class double
C{4} = [4; NaN]      class double
C{5} = [-Inf; 11]    class uint32 (-Inf converted to 0)
C{6} = [6; 12]       class double
```

Example 7 — Using Custom Empty Values and Comments

You have a file data.csv that contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```
Designate what should be treated as empty values and as comments. Read in all other values from the file:

```matlab
fid = fopen('data5.csv');
C = textscan(fid, '%s%n%n%n%n', 'delimiter', ',', ...
    'treatAsEmpty', {'NA', 'na'}, ...
    'commentStyle', '//');
fclose(fid);
```

This returns the following data in cell array C:

```matlab
C{:}
ans =
    'abc'
    'def'
ans =
    2
    NaN
ans =
    NaN
    5
ans =
    3
    6
ans =
    4
    7
```

**Example 8 — Reading From a String**

Read in a string (quoted from Albert Einstein) using `textscan`:

```matlab
str = ...
    ['Do not worry about your difficulties in Mathematics.' ...
    'I can assure you mine are still greater.'];
```

```matlab
s = textscan(str, '%s', 'delimiter', '.');
```

`s{:}`
ans =
    'Do not worry about your difficulties in Mathematics'
    'I can assure you mine are still greater'

**Example 9 — Handling Multiple Delimiters**

This example takes a comma-separated list of names, the test pilots known as the Mercury Seven, and uses `textscan` to return a list of their names in a cell array. When some names are removed from the input list, leaving multiple sequential delimiters, `textscan`, by default, accounts for this. If you override that default by calling `textscan` with the `multipleDelimsAsOne` option, `textscan` ignores the missing names.

Here is the full list of the astronauts:

```matlab
Mercury7 = ...
    'Shepard,Grissom,Glenn,Carpenter,Schirra,Cooper,Slayton';
```

Remove the names Grissom and Cooper from the input string, and `textscan`, by default, does not treat the multiple delimiters as one, and returns an empty string for each missing name:

```matlab
Mercury7 = 'Shepard,,Glenn,Carpenter,Schirra,,Slayton';
names = textscan(Mercury7, '%s', 'delimiter', ',');
names{:}'
ans =
    'Shepard' '' 'Glenn' 'Carpenter' 'Schirra' '' 'Slayton'
```

Using the same input string, but this time setting the `multipleDelimsAsOne` switch, `textscan` ignores the multiple delimiters:

```matlab
names = textscan(Mercury7, '%s', 'delimiter', ',', ...
    'multipledelimsasone', 1);
names{:}'
ans =
    'Shepard' 'Glenn' 'Carpenter' 'Schirra' 'Slayton'
```
Example 10 — Using the CollectOutput Switch

Shown below are the contents of a file wire_gage.txt. The first line contains four column headers in text. The lines that follow that are numeric data:

<table>
<thead>
<tr>
<th>AWG</th>
<th>Area</th>
<th>Resistance</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>211600</td>
<td>0.049</td>
<td>0.46</td>
</tr>
<tr>
<td>000</td>
<td>167810</td>
<td>0.0618</td>
<td>0.40965</td>
</tr>
<tr>
<td>00</td>
<td>133080</td>
<td>0.078</td>
<td>0.3648</td>
</tr>
<tr>
<td>0</td>
<td>105530</td>
<td>0.0983</td>
<td>0.32485</td>
</tr>
<tr>
<td>1</td>
<td>83694</td>
<td>0.124</td>
<td>0.2893</td>
</tr>
<tr>
<td>2</td>
<td>66373</td>
<td>0.1563</td>
<td>0.25763</td>
</tr>
<tr>
<td>3</td>
<td>52634</td>
<td>0.197</td>
<td>0.22942</td>
</tr>
<tr>
<td>4</td>
<td>41742</td>
<td>0.2485</td>
<td>0.20431</td>
</tr>
<tr>
<td>5</td>
<td>33102</td>
<td>0.3133</td>
<td>0.18194</td>
</tr>
<tr>
<td>6</td>
<td>26250</td>
<td>0.3951</td>
<td>0.16202</td>
</tr>
<tr>
<td>7</td>
<td>20816</td>
<td>0.4982</td>
<td>0.14428</td>
</tr>
<tr>
<td>8</td>
<td>16509</td>
<td>0.6282</td>
<td>0.12849</td>
</tr>
<tr>
<td>9</td>
<td>13094</td>
<td>0.7921</td>
<td>0.11443</td>
</tr>
<tr>
<td>10</td>
<td>10381</td>
<td>0.9989</td>
<td>0.10189</td>
</tr>
</tbody>
</table>

When you read the file with textscan having the CollectOutput switch set to zero, MATLAB returns each column of the numeric data in a separate 44-by-1 cell array:

```matlab
format long g
fid = fopen('wire_gage.txt', 'r');
C_text = textscan(fid, '%s', 4, 'delimiter', '|');
C_data0 = textscan(fid, '%d %f %f %f', 'CollectOutput', 0)
C_data0 =
    [44x1 int32]    [44x1 double]    [44x1 double]    [44x1 double]
```

Reading the file with CollectOutput set to one collects all data of a common type, double in this case, into a single 44-by-3 cell array:
frewind(fid)

C_text = textscan(fid, '%s', 4, 'delimiter', '|');

C_data1 = textscan(fid, '%d %f %f %f', 'CollectOutput', 1)
C_data1 =
    [44x1 int32]    [44x3 double]

See Also
dlmread, dlmwrite, xlswrite, fopen, fseek, importdata
textwrap

Purpose
Wrapped string matrix for given uicontrol

Syntax
outstring = textwrap(h,instring)
[outstring,position]=textwrap(h,instring)

Description
outstring = 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.

[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.

Example
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**
Measure performance using stopwatch timer

**Syntax**
```
tic
    any statements
toc
    t = toc
```

**Description**
tic starts a stopwatch timer.
toc prints the elapsed time since tic was used.
t = toc returns the elapsed time in t.

**Remarks**
The tic and toc functions work together to measure elapsed time. tic saves the current time that toc uses later to measure the elapsed time.
The sequence of commands
```
tic
    operations
toc
```
measures the amount of time MATLAB takes to complete one or more operations, and displays the time in seconds.

**Examples**
This example measures how the time required to solve a linear system varies with the order of a matrix.
```
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic
    x = A\b;
    t(n) = toc;
end
plot(t)
```

**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 2-3256 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.

```matlab
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>AveragePeriod</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 double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default NaN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only Always</td>
</tr>
<tr>
<td>BusyMode</td>
<td>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 'queue' — Execute function at next opportunity.</td>
<td>Data type Enumerated string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Values 'drop' 'error' 'queue'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default 'drop'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only While Running = 'on'</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Function that the timer executes when an error occurs. This function executes before the StopFcn. See “Creating Callback Functions” for more information.</td>
<td>Data type Text string, function handle, or cell array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Values: 'singleShot', 'fixedDelay', 'fixedRate', 'fixedSpacing'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: 'singleShot'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: NaN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only: Always</td>
</tr>
<tr>
<td>Name</td>
<td>User-supplied name.</td>
<td>Data type: Text string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: 'timer-(i)', where (i) is a number indicating the (i)th timer object created this session. To reset (i) to 1, execute the clear classes command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>Values: Any number &gt;= 0</td>
<td>Values: Any number &gt;= 0</td>
</tr>
<tr>
<td></td>
<td>Default: 0</td>
<td>Default: 0</td>
</tr>
<tr>
<td></td>
<td>Read only: While Running = 'on'</td>
<td>Read only: Never</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</td>
</tr>
<tr>
<td></td>
<td>Default: None</td>
<td>Default: None</td>
</tr>
<tr>
<td></td>
<td>Read only: Never</td>
<td>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>
</tbody>
</table>
| StopFcn       | Function the timer calls when it stops. The timer stops when  
|               | - You call the timer stop function  
|               | - The timer finishes executing TimerFcn, i.e., the value of TasksExecuted reaches the limit set by TasksToExecute.  
|               | - An error occurs (The ErrorFcn is called first, followed by the StopFcn.)  
|               | See “Creating Callback Functions” for more information. | Date type  
|               | Default None | Text string, function handle, or cell array  
|               | Read only Never | |
| Tag           | User supplied label. | Data type  
|               | Default Empty string ("") | Text string  
<p>|               | 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>TasksToExecute</td>
<td>Specifies the number of times the timer should execute the function specified in the TimerFcn property.</td>
<td>Data type double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Values Any number &gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only Never</td>
</tr>
<tr>
<td>TasksExecuted</td>
<td>The number of times the timer has called TimerFcn since the timer was started.</td>
<td>Data type double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Values Any number &gt;= 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only Always</td>
</tr>
<tr>
<td>TimerFcn</td>
<td>Timer callback function. See “Creating Callback Functions” for more information.</td>
<td>Data type Text string, function handle, or cell array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only Never</td>
</tr>
<tr>
<td>Type</td>
<td>Identifies the object type.</td>
<td>Data type Text string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Values 'timer'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only Always</td>
</tr>
<tr>
<td>UserData</td>
<td>User-supplied data.</td>
<td>Data type User-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default []</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read only Never</td>
</tr>
</tbody>
</table>
Purpose

Find timer objects

Syntax

- `out = timerfind`  
- `out = timerfind('P1', V1, 'P2', V2,...)`  
- `out = timerfind(S)`  
- `out = timerfind(obj, 'P1', V1, 'P2', V2,...)`

Description

`out = timerfind` returns an array, `out`, of all the timer objects that exist in memory.

`out = 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.

`out = 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.

`out = 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'.

2-3263
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'.

2-3265
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

ts = timeseries

ts = timeseries(Data)

ts = timeseries(Name)

ts = timeseries(Data,Time)

ts = timeseries(Data,Time,Quality)

ts = timeseries(Data,...,'Parameter',Value,...)

Description

ts = timeseries creates an empty time-series object.

ts = timeseries(Data) creates a time series with the specified Data.

ts has a default time vector that ranges from 0 to N-1 with a 1-second
intervals, where N is the number of samples. The default name of the
timeseries object is 'unnamed'.

ts = timeseries(Name) creates an empty time series with the name
specified by a string Name. This name can differ from the time-series
variable name.

ts = timeseries(Data,Time) 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.

ts = timeseries(Data,Time,Quality) 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.

ts = timeseries(Data,...,'Parameter',Value,...) creates a
timeseries object with optional parameter-value pairs after the
Data, Time, and Quality arguments. You can specify the following
parameters:

- Name — Time-series name entered as a string
- 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.
• `IsDatenum` — Logical value (`true` or `false`) that when set to `true` specifies that `Time` values are dates in the format of MATLAB serial dates.

**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, tndata.event, tsprops
title

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.

Examples
Display today's date in the current axes:

```
title(date)
```

Include a variable's value in a title:

```
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:

n = 3;
title([ 'Case number #',int2str(n)],'Color','y')

Include Greek symbols in a title:

title('\ite^\omega\tau = \cos(\omega\tau) + \isin(\omega\tau)')

Include a superscript character in a title:

title('\alpha^2')

Include a subscript character in a title:

title('X_1')

The text object String property lists the available symbols.

Create a multiline title using a multiline cell array.

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 String property for more information.

**See Also**

gtext, int2str, num2str, text, xlabel, ylabel, zlabel

“Annotating Plots” on page 1-86 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:

    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.

T = toeplitz(r) returns the symmetric or Hermitian Toeplitz matrix formed from vector r, where r defines the first row of the matrix.

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
**toolboxdir**

**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. The base directory of all toolboxes installed with MATLAB 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 MATLAB Compiler.

**Example**
To obtain the pathname for Control System Toolbox, run
```
s = toolboxdir('control')
```
MATLAB returns
```
s = \myhome\r2007a\matlab\toolbox\control
```

**See Also**
matlabroot
ctfroot in MATLAB Compiler
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  \( \text{trace} \) is a single-statement M-file.

\[
    t = \text{sum(diag(A))};
\]

See Also  \( \text{det}, \text{eig} \)
**Purpose**

Transpose timeseries object

**Syntax**

\[ \text{ts1} = \text{transpose} (\text{ts}) \]

**Description**

\( \text{ts1} = \text{transpose} (\text{ts}) \) returns a new timeseries object \( \text{ts1} \) with \( \text{IsTimeFirst} \) value set to the opposite of what it is for \( \text{ts} \). For example, if \( \text{ts} \) has the first data dimension aligned with the time vector, \( \text{ts1} \) 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 \( \text{transpose} (\text{ts.Data}) \) or \( (\text{ts.Data}).' \). 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 MATLAB displays the size for time-series data (up to three dimensions) 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

\( \text{trapz}(Y) \) 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.

\( \text{trapz}(X,Y) \) 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.

\( \text{trapz}(\ldots,\text{dim}) \) 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

\[ X = 0:\text{pi}/100:\text{pi}; \]
\[ Y = \sin(X); \]

Then both

\[ Z = \text{trapz}(X,Y) \]
and
\[ Z = \pi/100 \cdot \text{trapz}(Y) \]

produce
\[ Z = 1.9998 \]

**Example 2**
A nonuniformly spaced example is generated by

```matlab
X = sort(rand(1,101)*pi);
Y = sin(X);
Z = 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:

```matlab
z = exp(1i*pi*(0:100)/100);
trapz(z, 1./z)
```

```
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.)

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:

```matlab
nodes(1) = 0; % Root node
```

`nodes(2)` and `nodes(8)` are children of `nodes(1)`, so set these elements of the input vector to 1:

```matlab
nodes(2) = 1; nodes(8) = 1;
```

`nodes(5:7)` are children of `nodes(4)`, so set these elements to 4:

```matlab
nodes(5) = 4; nodes(6) = 4; 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:

```matlab
nodes = [0 1 2 2 4 4 1 8 8 10 10];
```

Now call `treeplot` to generate the plot:

```matlab
treeplot(nodes)
```

**See Also**

`etree`, `etreeplot`, `treelayout`
Purpose

Lower triangular part of matrix

Syntax

\[ L = \text{tril}(X) \]
\[ L = \text{tril}(X,k) \]

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

\[ \text{tril(ones(4,4),-1)} \]

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0
\end{bmatrix}
\]

See Also

diag, 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 \( \text{Tri} \) as a mesh. Each row of \( \text{Tri} \) defines a single triangular face by indexing into the vectors or matrices that contain the \( X, Y, \) and \( Z \) vertices.

trimesh(Tri,X,Y,Z,C) specifies color defined by \( C \) in the same manner as the \text{surf} function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

trimesh(...'PropertyName',PropertyValue...) specifies additional patch property names and values for the patch graphics object created by the function.

\( h = \text{trimesh}(...) \) returns a handle to a patch graphics object.

**Example**

Create vertex vectors and a face matrix, then create a triangular mesh plot.

\[
\begin{align*}
x &= \text{rand}(1,50); \\
y &= \text{rand}(1,50); \\
z &= \text{peaks}(6*x-3,6*x-3); \\
\text{tri} &= \text{delaunay}(x,y); \\
\text{trimesh}(&\text{tri},x,y,z)
\end{align*}
\]

**See Also**

patch, tetramesh, triplot, trisurf, delaunay

“Creating Surfaces and Meshes” on page 1-96 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 \( \int_{x_{\text{min}}}^{x_{\text{max}}} \int_{y_{\text{min}}}^{y_{\text{max}}} \int_{z_{\text{min}}}^{z_{\text{max}}} f(x,y,z) \, dz \, dy \, dx \) over the three dimensional rectangular region \( x_{\text{min}} \leq x \leq x_{\text{max}}, y_{\text{min}} \leq y \leq y_{\text{max}}, z_{\text{min}} \leq z \leq z_{\text{max}} \). 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.

“Parameterizing Functions Called by Function 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 \( 1.0 \times 10^{-6} \).

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 @integrand to triplequad:

\[
Q = \text{triplequad}(@\text{integrand}, 0, \pi, 0, 1, -1, 1);
\]

where the M-file integrnd.m is

\[
\text{function } f = \text{integrand}(x,y,z) \\
f = y \times \sin(x) + z \times \cos(x);
\]

Pass anonymous function handle \( F \) to triplequad:

\[
F = @(x,y,z)y \times \sin(x) + z \times \cos(x);
\]
Q = 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, quad, quad1, function handle (@), “Anonymous Functions”
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
Purpose
Triangular surface plot

Syntax
trisurf(Tri,X,Y,Z)
trisurf(Tri,X,Y,Z,C)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)

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. MATLAB 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

“Creating Surfaces and Meshes” on page 1-96 for related functions
**Purpose**

Upper triangular part of matrix

**Syntax**

\[ U = \text{triu}(X) \]
\[ U = \text{triu}(X,k) \]

**Description**

\[ U = \text{triu}(X) \] returns the upper triangular part of \( X \).

\[ U = \text{triu}(X,k) \] returns the element on and above 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.

![Diagram of upper triangular matrix](image)

**Examples**

\[ \text{triu}([1,0,0;0,1,0;0,0,1],-1) \]

\[ \text{ans} = \]

\[
\begin{array}{ccc}
1 & 1 & 1 \\
1 & 1 & 1 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
\end{array}
\]

**See Also**

diag, tril
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

**Description**

The general form of a `try` statement is

```
try,
    statement,
..., statement,
catch,
    statement,
..., statement,
end
```

Normally, only the statements between the `try` and `catch` are executed. However, if an error occurs during execution of any of the statements, the error is captured into `lasterror`, and the statements between the `catch` and `end` are executed. If an error occurs within the `catch` statements, execution stops unless caught by another `try...catch` block. The error string produced by a failed `try` block can be obtained with `lasterror`.

**See Also**

catch, rethrow, end, lasterror, eval, evalin
Purpose
Create tscollection object

Syntax
\[
\begin{align*}
tsc &= \text{tscollection}(\text{TimeSeries}) \\
tsc &= \text{tscollection}(\text{Time}) \\
tsc &= \text{tscollection}(\text{Time},\text{TimeSeries},'\text{Parameter}',\text{Value},...) \\
\end{align*}
\]

Description
\[
tsc = \text{tscollection}(\text{TimeSeries}) \text{ creates a tscollection object } tsc \text{ with one or more timeseries objects already in the MATLAB workspace. The argument TimeSeries can be a}
\]
- Single timeseries object
- Cell array of timeseries objects

\[
tsc = \text{tscollection}(\text{Time}) \text{ creates an empty tscollection object with the time vector Time. When time values are date strings, you must specify Time as a cell array of date strings.}
\]

\[
tsc = \text{tscollection}(\text{Time},\text{TimeSeries},'\text{Parameter}',\text{Value},...) \text{ creates a tscollection object with optional parameter-value pairs you enter after the Time and TimeSeries arguments. You can specify the following parameters:}
\]
- Name — String that specifies the name of this tscollection object
- IsDatenum — Logical value (true or false) that when set to true specifies that the Time values are dates in the format of MATLAB serial dates.

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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td><code>tscollection</code> name as a string. This can differ from the <code>tscollection</code> name in the MATLAB workspace.</td>
</tr>
<tr>
<td>Time</td>
<td>When <code>TimeInfo.StartDate</code> is empty, values are measured relative to 0. When <code>TimeInfo.StartDate</code> is defined, values represent date strings measured relative to the <code>StartDate</code>. The length of <code>Time</code> must be the same as the first or the last dimension of <code>Data</code> for each collection.</td>
</tr>
<tr>
<td>TimeInfo</td>
<td>Contains fields for contextual information about <code>Time</code>:</td>
</tr>
<tr>
<td></td>
<td>• Units — Time units with any of the following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', 'nanoseconds'</td>
</tr>
<tr>
<td></td>
<td>• Start — Start time</td>
</tr>
<tr>
<td></td>
<td>• End — End time (read only)</td>
</tr>
<tr>
<td></td>
<td>• Increment — Interval between subsequent time values. NaN when times are not uniformly sampled.</td>
</tr>
<tr>
<td></td>
<td>• Length — Length of the time vector (read only)</td>
</tr>
<tr>
<td></td>
<td>• Format — String defining the date string display format. See <code>datestr</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>StartDate</code> — Date string defining the reference date. See <code>setabstime</code> (<code>tscollection</code>).</td>
</tr>
<tr>
<td></td>
<td>• <code>UserData</code> — Any additional user-defined information</td>
</tr>
</tbody>
</table>
Examples

The following example shows how to create a `tscollection` object.

1 Import the sample data.

   ```
   load count.dat
   ```

2 Create three `timeseries` objects to store each set of data:

   ```
   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:

   ```
   tsc = tscollection({count1 count2},'name','tsc')
   ```

See Also

`addts`, `datestr`, `setabstime` (tscollection), `timeseries`, `tsprops`
Purpose

Construct event object for timeseries object

Syntax

\[
e = \text{tsdata.event}(\text{Name}, \text{Time})
\]
\[
e = \text{tsdata.event}(\text{Name}, \text{Time}, 'Datenum')
\]

Description

\(e = \text{tsdata.event}(\text{Name}, \text{Time})\) 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.

\(e = \text{tsdata.event}(\text{Name}, \text{Time}, 'Datenum')\) uses 'Datenum' to indicate that the Time value is a serial date number generated by the \text{datenum} function. The Time value is converted to a date string after the event is created.

Remarks

You add events by using the \text{addevent} method.

Fields of the \text{tsdata.event} object include the following:

- \text{EventData} — MATLAB array that stores any user-defined information about the event
- \text{Name} — String that specifies the name of the event
- \text{Time} — Time value when this event occurs, specified as a real number
- \text{Units} — Time units
- \text{StartDate} — A reference date, specified in MATLAB \text{datestr} format. \text{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 NaN for all points outside the convex hull. Requires a triangulation \( \text{TRI} \) of the points \( x,y \) obtained from \text{delaunay}.

**See Also**  
dealaunay, 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 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
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 TreatNaNsMissing property to determine how missing data is treated in calculations.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</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 <code>tsdata.interpolation</code> object that specifies the interpolation method for this time series. For more information, type <code>help tsdata.interpolation</code> at the MATLAB prompt. Fields of the <code>tsdata.interpolation</code> object include:</td>
</tr>
<tr>
<td></td>
<td>- <code>Fhandle</code> — Function handle to a user-defined interpolation function</td>
</tr>
<tr>
<td></td>
<td>- <code>Name</code> — 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>• <code>UserData</code> — Any user-defined information entered as a string</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| Events   | An array of `tsdata.event` objects that stores event information for this time series. You add events by using the `addevent` method. For more information, type `help tsdata.event` at the command line. Fields of the `tsdata.event` object include the following:

- **EventData** — 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 or a date string
- **Units** — Time units
- **StartDate** — A reference date specified in MATLAB date-string format. `StartDate` is empty when you have a numerical (non-date-string) time vector. |
<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>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>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<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>
### Property Description

**TimeInfo**

Uses the following fields for storing contextual information about **Time**:

- **Units** — Time units can have any of the following values: 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', or 'nanoseconds'
- **Start** — Start time
- **End** — End time (read only)
- **Increment** — Interval between two subsequent time values
- **Length** — Length of the time vector (read only)
- **Format** — String defining the date string display format. See the [MATLAB datestr function reference page](https://www.mathworks.com/help/techdoc/ref/datestr.html) for more information.
- **StartDate** — Date string defining the reference date. See the [MATLAB setabstime (timeseries) function reference page](https://www.mathworks.com/help/techdoc/ref/setabstime.html) for more information.
- **UserData** — Stores any additional user-defined information

**TreatNaNasMissing**

Logical value that specifies how to treat NaN values in Data:

- **true** — (Default) Treat all NaN values as missing data except during statistical calculations.
- **false** — Include NaN values in statistical calculations, in which case NaN values are propagated to the result.

### See Also

`datestr`, `get (timeseries)`, `set (timeseries)`, `setabstime (timeseries)`
Purpose
Open Time Series Tools GUI

Syntax
- `tstool`
- `tstool(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 partial pathname.

If you do not specify a filename 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
**Purpose**
Convert data types without changing underlying data

**Syntax**

```matlab
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 =
```

2-3307
Example 2

Set X to a 1-by-3 vector of 32-bit integers, then cast it to an 8-bit integer type:

```matlab
X = 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:

```matlab
Y = typecast(X, 'uint8')
Y =
    1   0   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):

```matlab
Y(9:12)
ans =
    0   1   0   0
```

Note that length(Y) is equal to 4.*length(X). Also note the difference between the output of typecast versus that of cast:

```matlab
Z = cast(X, 'uint8')
Z =
    1   255   255
```

Example 3

This example casts a smaller data type (uint8) into a larger one (uint16). Displaying the numbers in hexadecimal format makes it easier to see just how the data is being rearranged:

```matlab
format hex
X = uint8([44 55 66 77])
X =
```
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:

\[
Y = \text{typecast}(X, \text{'uint16'})
\]

\[
Y = \\
2c37 \quad 424d
\]

The second is done on a little-endian system. Note the difference in byte ordering:

\[
Y = \text{typecast}(X, \text{'uint16'})
\]

\[
Y = \\
372c \quad 4d42
\]

You can format the little-endian output into big-endian (and vice versa) using the \text{swapbytes} function:

\[
Y = \text{swapbytes(typecast(X, \text{'uint16'})})
\]

\[
Y = \\
2c37 \quad 424d
\]

**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:

```
format hex

typecast(uint8([120 86 52]), \text{'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

\[
typecast(uint8([120 86 52 18]), 'uint32')
\]
\[
ans =
12345678
\]

See Also

cast, class, swapbytes
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.

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.

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.

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.

handle = uibuttongroup(...) creates a uibuttongroup object and returns a handle to it in handle.

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.

**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.

% 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.

```matlab
function selcbk(source,eventdata)
    disp(source);
    disp([eventdata.EventName,' ',
         get(eventdata.OldValue,'String'),' ',
         get(eventdata.NewValue,'String')]);
    disp(get(get(source,'SelectedObject'),'String'));
```

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  

See Also  
uiControl, uipanel
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>UIContextMenu</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 `Visible` property of child axes, panels, and button groups. Does not affect child user interface controls (`uicontrol`).

**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** | **line**

  *Border of the uibuttongroup area.* Used to define the button group 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 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

```
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

```
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.
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 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

```
[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.

```matlab
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

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.
# Uibuttongroup Properties

<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'.

```matlab
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,
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, but still exists and you can query and set its properties.

---

**Note** The value of a uibuttongroup’s Visible property also controls the Visible property of child axes, uipanels, and uibuttongroups. This property does not affect the Visible property of child uicontrols.
### Purpose
Create context menu

### Syntax
```matlab
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.

```matlab
% Define the context menu
cmenu = uicontextmenu;
% Define the line and associate it with the context menu
hline = plot(1:10, 'UIContextMenu', cmenu);
% Define callbacks for context menu items
cb1 = ['set(hline, ''LineStyle'', ''--'')'];
cb2 = ['set(hline, ''LineStyle'', '':'')'];
cb3 = ['set(hline, ''LineStyle'', ''-'')'];
% Define the context menu items
item1 = uimenu(cmenu, 'Label', 'dashed', 'Callback', cb1);
item2 = uimenu(cmenu, 'Label', 'dotted', 'Callback', cb2);
item3 = uimenu(cmenu, 'Label', 'solid', 'Callback', cb3);
```

When the user right clicks or presses **Alt+click** on the line, the context menu appears, as shown in this figure:
See Also

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 uimenus 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

```matlab
set(0,'DefaultUiContextMenuCreateFcn','set(gcao,...
   ''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

```
hpt = uicontextmenu(...,'CreateFcn','set(gcao,...
    '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.

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**

*Uicontextmenu’s parent.* The handle of the uicontextmenu’s parent object. You can move a uicontextmenu object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

**Position**

*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**

*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 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 differentiate between single and double clicks and set the figure SelectionType property to normal or open accordingly before evaluating the list box’s Callback property.

- `'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

- 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.

```matlab
h = uicontrol('Style', 'pushbutton', 'String', 'Clear', ...
      'Position', [20 150 100 70], 'Callback', 'cla');
```

This statement gives focus to the pushbutton.

```matlab
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:

```matlab
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:

```matlab
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>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>
<tr>
<td>Position</td>
<td>Size and location of uicontrol object</td>
</tr>
</tbody>
</table>
**UiControl Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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>UIContextMenu</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.

**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:

- 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 **Ctl+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.

**Children**

matrix

The empty matrix; uicontrol objects have no children.

**Clipping**

{on} | off

This property has no effect on uicontrol objects.

**CreateFcn**

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.

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:

4 Sets the figure’s SelectionType property.
5 Sets the figure’s CurrentPoint property.
6 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:

\[ [0,0,\text{width},\text{height}] \]

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 Extent is returned as a single line, even if the string wraps when displayed on the control.

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.

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.

FontSize
size in FontUnits
**Uicontrol Properties**

*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.

**ForeColor**

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 MATLAB’s 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 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
**Uicontrol Properties**

*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

*KeyPressFcn* is 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 structure argument.

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Description</th>
<th>Examples:</th>
</tr>
</thead>
</table>
| Character                  | Character interpretation of the key that was pressed. | 'a'  
|                           |                                                  | '='  
|                           |                                                  | ' '   
|                           |                                                  | 'A'   |
| Modifier                   | Current modifier, such as 'control', or an empty cell array if there is no modifier | {1x0 cell}  
|                           |                                                  | {1x0 cell}  
|                           |                                                  | {'shift'}  
|                           |                                                  | {'shift'}  |
| Key                        | Name of the key that was pressed.                 | 'a'  
|                           |                                                  | 'equal'  
|                           |                                                  | 'shift'   
|                           |                                                  | 'a'   |

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:
**Uicontrol Properties**

- **Check boxes** – Max is the setting of the Value property while the check box is selected.

- **Editable text** – If Max - Min > 1, then editable text boxes accept multiline input. If Max - Min <= 1, then editable text boxes accept only single line input.

- **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.

- **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** – If Max - Min > 1, then editable text boxes accept multiline input. If Max - Min <= 1, then editable text boxes accept only single line input.

- **List boxes** – If Max - Min > 1, then list boxes allow multiple item selection. If Max - Min <= 1, then list boxes allow only single item selection.
Uicontrol Properties

- Radio buttons – Min is the setting of the Value property when the radio button is not selected.
- Sliders – Min is the minimum slider value and must be less than Max. The default is 0.
- Toggle buttons – Min is the value of the 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 Min property.

Parent handle

*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.

Position position rectangle

*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 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 uicontrol object. width and height are the dimensions of the uicontrol rectangle. All measurements are in units specified by the 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 \([0, 1]\). 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.

For example, if you create the following slider,

```matlab
uicontrol('Style','slider','Min',1,'Max',7,...
      'Value',2,'SliderStep',[0.1 0.6])
```

clicking on the arrow button moves the indicator by,

```
0.1*(7-1)
ans =
   0.6000
```

and clicking in the trough moves the indicator by,

```
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><strong>String Property Format</strong></th>
<th><strong>Example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell array of strings</td>
<td>{'one' 'two' 'three'}</td>
</tr>
<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 editable text, the String property value is set to the string entered by the user.

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,

```matlab
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).
UiControl Properties

- 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 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

_Uicontrol visibility._ By default, all uicontrols are visible. When set to off, the uicontrol is not visible, but still exists and you can query and set its properties.

**Note** Setting Visible to off for uicontrols that are not displayed initially in the GUI, can result in faster startup time for the GUI.
**Purpose**

Open standard dialog box for selecting a 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 Windows platforms, `uigetdir` opens a dialog box in the base directory (the Windows desktop) with the current directory selected. See “Remarks” on page 2-3373 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 2-3373 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 2-3373 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 2-3374 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_title` string is similar to the one shown in the following figure.

![Select Directory to Open](image)
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](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:

\[
\text{uigetdir(matlabroot,'MATLAB Root Directory')}
\]
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, uigetdir returns 0.

Note A modal dialog box prevents the user from interacting with other
windows before responding. To block MATLAB program execution, use
the uiwait 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.
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. A
FilterSpec string can also be a filename. In this case the filename
becomes the default filename and the file's extension is used as the
default filter. If FilterSpec is a string, uigetfile appends 'All
Files' to the list of file types.

If FilterSpec is a cell array, 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 2-3381 and “Example 3” on page 2-3382 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.

```matlab
[FileName, PathName, FilterIndex] = uigetfile(FilterSpec, DialogTitle)
```

**Note** For Mac platforms, `DialogTitle` is ignored.

```matlab
[FileName, PathName, FilterIndex] = uigetfile(FilterSpec, DialogTitle, DefaultName)
```

`uigetfile` displays a dialog box that has the title `DialogTitle`. To use the default file types and specify a dialog title, enter

```matlab
uigetfile('', DialogTitle)
```

In this case, `uigetfile` opens the dialog box in the directory specified by the path. See “Example 6” on page 2-3385. If the path does not include a filename, it must end with a slash (/) or backslash (\) separator. For example, `'C:\Work\'`. Note that `uigetfile` recognizes both './' and '../' as valid values. If the specified path does not exist, `uigetfile` opens the dialog box in the current directory.
[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',selectmode) sets the multiselect mode to specify if multiple file selection is enabled for the uigetfile dialog. Valid values for selectmode are 'on' and 'off' (default). If 'MultiSelect' is 'on' and the user selects more than one file in the dialog box, then 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, PathName is always a string that represents a single directory.

Remarks

For 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 2-3380 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.

```matlab
[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.

```matlab
[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.

```
[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.

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('*.m', '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.

    uigetfile({ '*.jpg;*.tif;*.png;*.gif','All Image Files';...
       '*. ','All Files' },'mytitle',...
       'C:\Work\myfile.jpg')
uigetfile

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'.

2-3387
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.

![Closing Figure](2-3389)

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', ...
    'savefigurebeforeclosing', ...
    'Closing Figure', ...
    {'Do you want to save your figure before closing?'
```
\begin{verbatim}
'You can save your figure manually by typing \texttt{``hgsave(gcf)''}, ... 
\{\texttt{``always''}, \texttt{``never''}; \texttt{``Yes''}, \texttt{``No''}\}, ... % Values and button strings
\texttt{``ExtraOptions''}, \texttt{``Cancel''}, ... % Additional button
\texttt{``DefaultButton''}, \texttt{``Cancel''}, ... % Default choice
\texttt{``HelpString''}, \texttt{``Help''}, ... % String for Help button
\texttt{``HelpFcn''}, \texttt{doc\texttt{\{``closereq''\})} % Callback for Help button
\end{verbatim}

\textbf{See Also} \hspace{1cm} addpref, getpref, ispref, rmpref, setpref, uisetpref
uiimport

**Purpose**
Open Import Wizard to import data

**Syntax**
```matlab
uiimport
uiimport(filename)
uiimport('-file')
uiimport('-pastespecial')
S = uiimport(...)```

**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 = uiimport(...) 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, clipboard
uimenu

**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 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.

Uimenus 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 uimenus on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user).
When MenuBar is none, uimenu items are 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:

```matlab
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.
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
```
Uimenu Properties

```matlab
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 uimenus are other uimenus, 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 uimenus. This default applies unless you override it by specifying a different CreateFcn callback when you call uimenu. For example, the code
Uimenu Properties

```
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.

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 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
go

{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

'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.
**Uimenu 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 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.

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 = 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:

```matlab
intmin('uint32')
ans =
  0
intmax('uint32')
ans =
4294967295
```

**See Also**

double, single, int8, int16, int32, int64, intmax, intmin
**Purpose**

Open file selection dialog box with appropriate file filters

**Syntax**

```matlab
uiopen
uiopen('MATLAB')
uiope('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**

`uiopen` cannot be compiled. If you want to create a file selection dialog that can be compiled, use `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)**:

![Open dialog box](image)

**See Also**

`uigetfile`, `uiputfile`, `uisave`
Purpose

Create panel container object

Syntax

\[ h = \text{uipanel}('PropertyName1',\text{value1},'PropertyName2',\text{value2}, \ldots) \]

\[ h = \text{uipanel}(\text{parent},'PropertyName1',\text{value1},'PropertyName2', \text{value2}, \ldots) \]

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.

\[ h = \text{uipanel}('PropertyName1',\text{value1},'PropertyName2',\text{value2}, \ldots) \]
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.

\[ h = \text{uipanel}(\text{parent},'PropertyName1',\text{value1},'PropertyName2',\text{value2}, \ldots) \]
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.

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]);
```

**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>
</tbody>
</table>
### Uipanel Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 `Visible` property of child axes, uibuttongroups, and uipanels. Does not affect child uicontrols.

**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

*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.
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).

```
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 (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 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

\[ \text{[left bottom width height]} \]

left and bottom are the distance from the lower-left corner of the parent object to the lower-left corner of the 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.

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');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

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.

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.** 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, but still exists and you can query and set its properties.

---

**Note** The value of a uipanel’s Visible property also controls the Visible property of child axes, uipanels, and uibuttongroups. This property does not affect the Visible property of child uicontrols.
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 uitoolbar 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 uitoolbar 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 uitoolbar and its uipushtool children is changed to modal, the uipushtools still exist and are contained in the Children list of the uitoolbar, but are not displayed until the figure WindowStyle is changed to normal or docked.

Examples
This example creates a uitoolbar object and places a uipushtool object on it.

h = figure('ToolBar','none')
ht = uitoolbar(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, uitoobar
Describe push tool 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:

```
set(h,'DefaultUipushtoolPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, a uitoolbar handle, or a uipushtool handle. PropertyName is the name of the Uipushtool 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.

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>
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, 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.`* 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

```plaintext
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

```plaintext
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`.

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.
Uipushtool Properties

- **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:

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 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.

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(uuitoolbarhandles,'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'.

**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 Properties**

*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.

![Save file name dialog box](image)

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.

**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.
[FileName, PathName, FilterIndex] = uiputfile(FilterSpec)
displays only those files with extensions that match FilterSpec.
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. A
FilterSpec string can also be a filename. In this case the filename
becomes the default filename and the file's extension is used as the
default filter. If FilterSpec is a string, uiputfile appends 'All
Files' to the list of file types.

If FilterSpec is a cell array, 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
3” on page 2-3444 and “Example 4” on page 2-3445 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

    uiputfile('', DIALOGTitle)

[FileName, PathName, FilterIndex] =
uiputfile(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 2-3447. If the path does not include a filename, it must end with a slash (/) or backslash (\) separator. For example, ‘C:\Work\’. Note that **uiputfile** recognizes both ‘./’ and ‘../’ as valid values. If the specified path does not exist, **uiputfile** opens the dialog box in the current directory.

**Remarks**

For 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.

```
[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'; '*.*'}, ...
    '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.

Example 5

The following code checks for the existence of the file and displays a message about the result of the open operation.

```
[filename, pathname] = uiputfile('*.m', 'Pick an M-file');
if isequal(filename, 0) || isequal(pathname, 0)
    disp('User selected Cancel')
else
```

2-3446
disp(['User selected',fullfile(pathname,filename)])
end

Example 6

tputfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
    '*.*','All Files '},'Save Image',...
    'C:\Work\newfile.jpg')

See Also

tgetdir, tgetfile
**Purpose**  
Control program execution

**Syntax**  
uiwait
uiwait(h)
uiwait(h,timeout)
uiresume(h)

**Description**  
The `uiwait` and `uiresume` functions block and resume MATLAB program execution.

- `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.
- `uiresume(h)` resumes the M-file execution that `uiwait` suspended.

**Remarks**  
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 `uiwait` function blocks execution.

- `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/uiresume` 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`.
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);

gcf is the handle of the figure that contains the object whose callback is executing.

“Using a Modal Dialog 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**
uicontrol, uimenu, waitfor, figure, dialog
**Purpose**
Open standard dialog box for saving workspace variables

**Syntax**

```
uisave
uisave(variables)
uisave(variables,filename)
```

**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.

![Save Workspace Variables dialog box](image)

**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.

**Note** uisave cannot be compiled. If you want to create a dialog that can be compiled, use uiputfile.

**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*.

\[
\begin{align*}
h &= 365; \\
g &= 52; \\
\text{uisave} &\{\text{'}h\text{'},\text{'}g\text{'},\text{'}var1\text{'}\};
\end{align*}
\]

![Save Workspace Variables dialog box](image)

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 = uisetcolor
```
```
c = uisetcolor([r g b])
```
```
c = uisetcolor(h)
```
```
c = uisetcolor(...,'dialogTitle')
```

**Description**

`c = 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 = 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 = 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 = uisetcolor(...,'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`
uisetfont

**Purpose**
Open standard dialog box for setting object’s font characteristics

**Syntax**

- `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
**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**  
```matlab
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.

```matlab
v = allchild(hObject)
uistack(v(3),'down',2)
```
Purpose
Create toggle button on toolbar

Syntax

```matlab
htt = uitoggletool('PropertyName1', value1, 'PropertyName2', value2,...)
htt = uitoggletool(ht,...)
```

Description

```matlab
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.

```matlab
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 tool bar 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 `uitoolbar` object and places a `uitoggletool` object on it.

```matlab
h = figure('ToolBar','none');
ht = uitoolbar(h);
a = rand(16,16,3);
```
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello');

See Also

get, set, uicontrol, uipushtool, uitoolbar
Uitoggletool Properties

**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:

\[
\text{set}(h, 'DefaultUitoggletoolProperty\text{PropertyName}', \text{PropertyValue}...)
\]

Where \( h \) can be the root handle (0), a figure handle, a uitoolbar handle, or a uitoggletool handle. \( \text{PropertyName} \) is the name of the Uitoggletool 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”.

**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>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>TooltipString</td>
<td>Content of object’s tooltip.</td>
</tr>
<tr>
<td>Type</td>
<td>Object class.</td>
</tr>
<tr>
<td>UserData</td>
<td>User specified data.</td>
</tr>
<tr>
<td>Visible</td>
<td>Uitoggletool 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 sets the BeingDeleted property to on when the object’s delete function callback is called.
Uitoggletool Properties

(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'))
```

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.

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
Uitoggletool 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.

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 interrupted 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'.

```matlab
h = findobj(uuitoolbarhandles, 'Tag', 'Bold')
```

TooltipString
string

*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'.

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)
```
uitoolbar

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 `uitoolbar` 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 `uitoolbar` 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 uitooolbar'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>

For more information about the `uitoolbar` objects and their properties, see the documentation for `uitoolbar`.


## 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>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>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 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 uitoolbar object, in the order in which they appear on the toolbar. The children objects of uitoolbars are `uipushtools` and `uitoggletools`. 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 uitoolbar object. MATLAB sets all property values for the uitoolbar 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 uitoolbar object has no effect.

You can define a default `CreateFcn` callback for all new uitoolbars. This default applies unless you override it by specifying a different
CreateFcn callback when you call uitoolbar. 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 uitoolbar with a call similar to

```matlab
ht = uitoolbar(...,'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 uitoolbar 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.

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 uitoolbar object is deleted (e.g., when you call the delete function or cause the figure containing the uitoolbar 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.
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**

*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**

*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'.

```matlab
h = findobj(figurehandles,'Tag','FormatTb')
```

**Type**

*string (read-only)*

Object class. This property identifies the kind of graphics object. For uitoobar objects, Type is always the string 'uitoolbar'.
**Uitoolbar Properties**

**UserData**

array

*User specified data.* You can specify UserData as any array you want to associate with the uitoolbar 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.
Purpose

Undo previous checkout from source control system (UNIX)

GUI

As an alternative to the undocheckout function, select Source Control > Undo Checkout in the File menu of the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser. For more information, see “Undoing the Checkout”.

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 pathname 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 pathnames for filenames 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 Windows platforms, use verctrl.
**Purpose**  
Convert Unicode characters to numeric bytes

**Syntax**

```
bytes = unicode2native(unicodestr)
bytes = unicode2native(unicodestr, encoding)
```

**Description**

`bytes = unicode2native(unicodestr)` takes a char vector of Unicode characters, `unicodestr`, converts it to MATLAB's 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 ('''), MATLAB's default encoding scheme is used.

**Examples**

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.

```
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`
**Purpose**  
Find set union of two vectors

**Syntax**

c = union(A, B)  
c = union(A, B, 'rows')  
[c, ia, ib] = union(...)

**Description**
c = 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 = 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.

[c, ia, ib] = union(...) also returns index vectors ia and ib such that c = a(ia) \( \cup \) b(ib), or for row combinations, c = a(ia,:) \( \cup \) b(ib,:). If a value appears in both a and b, union indexes its occurrence in b. If a value appears more than once in b or in a (but not in b), 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, ia, ib] = union(a, b);  
c = [-1 0 1 2 3 4 6]  
ia = [3 4 5]  
ib = [2 3 4 8]
### union

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**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 is either
'first' or 'last', returns index vectors m and n such that

- The elements of vector m are the lowest indices of unique elements
  in A when occurrence is the string 'first' and the highest such
  indices when occurrence is 'last'.

- The elements of vector n are the lowest indices of unique elements
  in b when occurrence is the string 'first' and the highest such
  indices when occurrence is 'last'.

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

A = [1 1 5 6 2 3 3 9 8 6 2 4]
A =
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:

```matlab
[b1, m1, n1] = unique(A, 'first')
b1 =
    1 2 3 4 5 6 8 9
m1 =
    1 5 6 12 3 4 9 8
n1 =
    1 1 5 6 2 3 3 8 7 6 2 4
```

Verify that \( b_1 = A(m_1) \) and \( A = b_1(n_1) \):

```matlab
all(b1 == A(m1)) && 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:

```matlab
[b2, m2, n2] = unique(A, 'last')
b2 =
    1 2 3 4 5 6 8 9
m2 =
    2 11 7 12 3 10 9 8
n2 =
    1 1 5 6 2 3 3 8 7 6 2 4
```

Verify that \( b_2 = A(m_2) \) and \( A = b_2(n_2) \):

```matlab
all(b2 == A(m2)) && all(A == b2(n2))
ans =
    1
```
Because NaNs are not equal to each other, unique treats them as unique elements.

```matlab
unique([1 1 NaN NaN])
ans =
    1    NaN    NaN
```

**See Also**
intersect, ismember, issorted, setdiff, setxor, union
Purpose
Execute UNIX command and return result

Syntax
unix command
status = unix('command')
[status, result] = unix('command')
[status, result] = unix('command', '-echo')

Description
unix command calls upon the UNIX operating system to execute the given command.

status = unix('command') returns completion status to the status variable.

[status, result] = unix('command') returns the standard output to the result variable, in addition to completion status.

[status, result] = unix('command', '-echo') displays the results in the Command Window as it executes, and assigns the results to w.

Note MATLAB 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.

[s, w] = unix('who');

MATLAB returns 0 (success) in s and a string containing the list of users in w.

In this example

[s, w] = unix('why')
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.

See Also
dos, ! (exclamation point), perl, system

“Running External Programs” in the MATLAB Desktop Tools and Development Environment documentation
## Unload Library from Memory

### Purpose
Unload external library from memory

### Syntax
- `unloadlibrary('libname')`
- `unloadlibrary libname`

### Description
`unloadlibrary('libname')` unloads the functions defined in shared library `shrlib` from memory. If you need to use these functions again, you must first load them back into memory using `loadlibrary`.

`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, `shrlibsmaple`. Call one of its functions, and then unload the library:

```matlab
addpath([matlabroot '\extern\examples\shrlib'])
loadlibrary shrlibsmaple shrlibsmaple.h

s.p1 = 476;  s.p2 = -299;  s.p3 = 1000;
calllib('shrlibsmaple', 'addStructFields', s)
ans =
    1177

unloadlibrary shrlibsmaple
```

### See Also
- `loadlibrary`
- `libisloaded`
- `libfunctions`
- `libfunctionsview`
- `libpointer`
- `libstruct`
- `calllib`
Purpose

Piecewise polynomial details

Syntax

\[ \text{[breaks,coefs,l,k,d] = unmkpp(pp)} \]

Description

\[ \text{[breaks,coefs,l,k,d] = unmkpp(pp)} \]
extracts, from the piecewise polynomial pp, its breaks breaks, coefficients coefs, number of pieces l, order k, and dimension d of its target. Create pp using spline or the spline utility 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.

\[
\text{pp = mkpp([-8 -4],[-1/4 1 0])};
\text{[breaks,coefs,l,k,d] = unmkpp(pp)}
\]

\[
\text{breaks =}
\begin{align*}
-8 & -4 \\
\end{align*}
\]

\[
\text{coefs =}
\begin{align*}
-0.2500 & 1.0000 & 0 \\
\end{align*}
\]

\[
\text{l =}
\begin{align*}
1 \\
\end{align*}
\]

\[
\text{k =}
\begin{align*}
3 \\
\end{align*}
\]

\[
\text{d =}
\begin{align*}
1 \\
\end{align*}
\]

See Also

mkpp, ppval, spline
unregisterallevents

**Purpose**
Unregister all events for control

**Syntax**

```matlab
h.unregisterallevents
unregisterallevents(h)
```

**Description**
`h.unregisterallevents` unregisters all events that have previously been registered with control, `h`. After calling `unregisterallevents`, the control will no longer respond to any events until you register them again using the `registerevent` function.

`unregisterallevents(h)` is an alternate syntax for the same operation.

**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'; 'DblClick' 'my2click'; ...
     'MouseDown' 'mymoused'});

h.eventlisteners
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:

```matlab
h.unregisterallevents;
h.eventlisteners
an =
```
To unregister specific events, use the unregisterevent function. First, create the control and register three events:

```matlab
f = figure ('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f,...
    {'Click' 'myclick'; 'DblClick' 'my2click'; ...
     'MouseDown' 'mymoused'});
```

Next, unregister two of the three events. The mousedown event remains registered:

```matlab
h.unregisterevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});
```

**Excel Example**

Create an Excel Workbook object and register some events.

```matlab
excel = actxserver('Excel.Application');
wbs = excel.Workbooks;
wb = wbs.Add;
wb.registerevent({'Activate' 'EvtActivateHndlr'; ...
    'Deactivate' 'EvtDeactivateHndlr'})
```

MATLAB shows the events registered to their corresponding event handlers.

```matlab
ans =
    'Activate'    'EvtActivateHndlr'
    'Deactivate'  'EvtDeactivateHndlr'
```
unregisterallevents

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 =
{}
```

See Also

events, eventlisteners, registerevent, unregisterevent, isevent
**Purpose**
Unregister event handler with control’s event

**Syntax**

- `h.unregisterevent(event_handler)`
- `unregisterevent(h, event_handler)`

**Description**

- `h.unregisterevent(event_handler)` unregisters certain event handler routines with their corresponding events. Once you unregister an event, the control no longer responds to any further occurrences of the event.

- `unregisterevent(h, event_handler)` is an alternate syntax for the same operation.

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,

```matlab
h.unregisterevent({'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 `registerevent`, `unregisterevent` does not accept numeric event identifiers.

**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`:

```matlab
f = figure ('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ... 
    [0 0 200 200], f, ... 
    'sampev');

h.eventlisteners
ans =
```
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
ans =
    'click'    'sampev'
    'mousedown' '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
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:

```matlab
h.unregisterevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});

h.eventlisteners
ans =
    {}
In this last example, you could have used unregisterallevents instead:

```matlab
h.unregisterallevents;
```

**Excel Example**

Create an Excel Workbook object

```matlab
e = actxserver('Excel.Application');
wbs = e.Workbooks;
wb = wbs.Add;
```

Register two events with the your event handler routines, EvtActivateHndlr and EvtDeactivateHndlr.

```matlab
wb.registerevent({'Activate' 'EvtActivateHndlr'; ... 'Deactivate' 'EvtDeactivateHndlr'})
wb.eventlisteners
```

MATLAB shows the events with the corresponding event handlers.

```matlab
ans =

    'Activate'    'EvtActivateHndlr'
    'Deactivate'  'EvtDeactivateHndlr'
```

Next, unregister the Deactivate event handler.

```matlab
wb.unregisterevent({'Deactivate' 'EvtDeactivateHndlr'})
wb.eventlisteners
```

MATLAB shows the remaining registered event (Activate) with its corresponding event handler.

```matlab
ans =

    'Activate'    'EvtActivateHndlr'
```
unregisterevent

See Also

events, 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, files from rerunning untar on the same tar filename do not overwrite any of those files that have a read-only attribute; instead, untar issues a warning for such files. On Windows platforms, the hidden, system, and archive attributes are not set.

tarfilename 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 until a file exists. 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. outputdir is created if it does not exist.

untar(url, ...) extracts the tar archive from an Internet URL. The URL must include the protocol type (e.g., 'http://' or 'ftp://'). The URL is downloaded to a temporary directory and deleted.

filenames = untar(...) extracts the tar archive and returns the relative pathnames of the extracted files into the string cell array filenames.

**Examples**

Copy all .m files in the current directory to the directory backup:

```
tar('mymfiles.tar.gz','*.m');
untar('mymfiles','backup');
```
Run `untar` to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`:

```matlab
doc untar
url = 'http://www.mathworks.com/moler/ncm.tar.gz';
cmFiles = untar(url,'ncm')
```

**See Also**
- gzip, gunzip, tar, unzip, zip
**Purpose**
Correct phase angles to produce smoother phase plots

**Syntax**

```matlab
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.

```matlab
semilogx(w,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 2.6959
     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.

\[
Q = \\
0 & 7.0686 & 1.5708 & 2.3562 \\
0.1963 & 7.2649 & 1.7671 & 2.5525 \\
0.3927 & 7.4613 & 1.9635 & 2.7489 \\
0.5890 & 7.6576 & 2.1598 & 2.9452
\]

**See Also**

abs, angle
unzip

**Purpose**
Extract contents of zip file

**Syntax**

```matlab
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. 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 (e.g., `http://`). The URL is downloaded to the temp directory and deleted.

`filenames = unzip(...)` extracts the zip archive and returns the relative pathnames of the extracted files into the string cell array `filenames`.

`unzip` does not support password-protected or encrypted zip archives.

**Examples**

**Example 1**

Copy the demos HTML files to the directory archive:

```matlab
% 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')

Example 2
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 = upper('str')
B = upper(A)

**Description**  
t = upper('str') converts any lowercase characters in the string `str` to the corresponding uppercase characters and leaves all other characters unchanged.

B = upper(A) when A is a cell array of strings, returns a cell array the same size as A containing the result of applying `upper` to each string within A.

**Examples**  
upper('attention!') is ATTENTION!.

**Remarks**  
Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

**See Also**  
lower
Purpose

Read content at URL

Syntax

s = urlread('url')
s = urlread('url','method','params')
[s,status] = urlread(...)  

Description

s = urlread('url') reads the content at a URL into the string s. If the server returns binary data, s will be unreadable.

s = 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,status] = 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 server address and port. 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:

```
urlstring = sprintf('%s%s', ...
   'http://www.mathworks.com/matlabcentral/', ...
   'fileexchange/loadAuthorIndex.do');

s = urlread(urlstring);
```

Download Content from File on FTP Server

```
s=urlread(page);
```
urlread

MATLAB displays

s = urlread('file:///c:/winnt/matlab.ini')

See Also

urlwrite
tcpip if the Instrument Control Toolbox is installed
**Purpose**  
Save contents of URL to file

**Syntax**

```matlab
urlwrite('url','filename')
f = urlwrite('url','filename')
f = urlwrite('url','method','params')
[f,status] = 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 = urlwrite('url','filename')` reads the contents of the specified URL, saving the contents to `filename` and assigning `filename` to `f`.

`f = 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,status] = 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 server address and port. 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.

```matlab
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange
/Category.jsp?type=category&id=1','samples.html');
```

View the file in the Help browser.

```matlab
open('samples.html')
```
urlwrite

See Also urlread
Purpose
Determine whether Java feature is supported in MATLAB

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&lt;sup&gt;1&lt;/sup&gt; 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 is running</td>
</tr>
<tr>
<td>'swing'</td>
<td>Swing components&lt;sup&gt;2&lt;/sup&gt; are available</td>
</tr>
</tbody>
</table>

1. Java's GUI components in the Abstract Window Toolkit
2. Java's 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.

```matlab
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 a JVM.

```matlab
if ~usejava('jvm')
    error([mfilename ' requires Java to run.']);
end
```
See Also
javachk
Purpose  Vandermonde matrix

Syntax  \( A = \text{vander}(v) \)

Description  \( A = \text{vander}(v) \) returns the Vandermonde matrix whose columns are powers of the vector \( v \), that is, \( A(i,j) = v(i)^{(n-j)} \), where \( n = \text{length}(v) \).

Examples  \( \text{vander}(1:.5:3) \)

\[
\begin{array}{ccccc}
1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \\
5.0625 & 3.3750 & 2.2500 & 1.5000 & 1.0000 \\
16.0000 & 8.0000 & 4.0000 & 2.0000 & 1.0000 \\
39.0625 & 15.6250 & 6.2500 & 2.5000 & 1.0000 \\
81.0000 & 27.0000 & 9.0000 & 3.0000 & 1.0000 \\
\end{array}
\]

See Also  gallery
Purpose
Variance

Syntax
V = var(X)
V = var(X,1)
V = var(X,w)
V = var(X,w,dim)

Description
V = var(X) returns the variance of X for vectors. For matrices, var(X) is a row vector containing the variance of each column of X. For N-dimensional arrays, 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.

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 = var(X,1) normalizes by N and produces the second moment of the sample about its mean. var(X,0) is equivalent to var(X).

V = var(X,w) computes the variance using the weight vector w. The length of w must equal the length of the dimension over which var operates, and its elements must be nonnegative. The elements of w must be positive. var normalizes w to sum of 1.

V = var(X,w,dim) takes the variance along the dimension 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
Purpose

Variance of timeseries data

Syntax

\[
\text{ts\_var} = \text{var}(\text{ts})
\]
\[
\text{ts\_var} = \text{var}(\text{ts}, 'PropertyName1', \text{PropertyValue1}, ...)
\]

Description

\( \text{ts\_var} = \text{var}(\text{ts}) \) returns the variance of \( \text{ts.data} \). When \( \text{ts.Data} \) is a vector, \( \text{ts\_var} \) is the variance of \( \text{ts.Data} \) values. When \( \text{ts.Data} \) is a matrix, \( \text{ts\_var} \) is a row vector containing the variance 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, \( \text{var} \) always operates along the first nonsingleton dimension of \( \text{ts.Data} \).

\( \text{ts\_var} = \text{var}(\text{ts}, 'PropertyName1', \text{PropertyValue1}, ...) \) 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 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.

   \[
   \text{load count.dat}
   \]

2 Create a timeseries object with 24 time values.

   \[
   \text{count\_ts} = \text{timeseries(count,[1:24],'Name','CountPerSecond')}
   \]
3 Calculate the variance of each data column for this timeseries object.

```matlab
var(count_ts)
ans =

1.0e+003 *

0.6437   1.7144   4.6278
```

The variance is calculated independently for each data column in the timeseries object.

**See Also**

iqr (timeseries), mean (timeseries), median (timeseries), std (timeseries), timeseries
Purpose
Variable length input argument list

Syntax
function y = bar(varargin)

Description
function y = bar(varargin) accepts a variable number of arguments into function bar.m.

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
The function

    function myplot(x,varargin)
    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

    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

**Graphical Interface**

As an alternative to the `ver` function, select **About** from the **Help** menu in any product that has a **Help** menu.

**Syntax**

```matlab
ver
ver product
v = ver('product')
```

**Description**

`ver` displays a header containing the current version number, license number, operating system, and Java VM version for MATLAB, followed by the version numbers for Simulink, if installed, and all other MathWorks products installed.

`ver product` displays the MATLAB 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 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

```
% 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-2002` instead of `2-Sep-2002`.

**Examples**

Return version information for the Control System Toolbox by typing

```matlab
ver control
```

MATLAB returns
Return version information for the Control System Toolbox in a structure array, v.

\[
v = \text{ver('control')}
\]

\[
v =
\begin{align*}
&\text{Name: 'Control System Toolbox'} \\
&\text{Version: '7.1'} \\
&\text{Release: '(R2006b)'} \\
&\text{Date: '19-Sep-2006'}
\end{align*}
\]

Display version information on MathWorks 'Real-Time' products:

\[
v = \text{ver};
\text{for } k=1:\text{length}(v)
\quad \text{if } \text{strfind}(v(k).\text{Name}, \text{'Real-Time'})
\quad \text{disp}('\text{sprintf(''\%s, Version \%s'', }\ldots
\quad \quad \quad \quad \quad v(k).\text{Name}, v(k).\text{Version}))
\quad \text{end}
\text{end}
\]

Real-Time Windows Target, Version 2.6.2
Real-Time Workshop, Version 6.5
Real-Time Workshop Embedded Coder, Version 4.5

**See Also**

help, hostid, license, version, whatsnew

**Help > Check for Updates** in the MATLAB desktop.
**Purpose**

Source control actions (Windows)

**GUI Alternatives**

As an alternative to the `verctrl` function, use **Source Control** in the **File** menu of the Editor/Debugger, Simulink, or Stateflow, or in the context menu of the Current Directory browser.

**Syntax**

```
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 Windows Platforms 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/Debugger 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.

```matlab
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.

```matlab
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.

```matlab
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

   ??? 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 Windows Platforms topic 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 MATLAB.

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 desired toolbox and use that directory name. To see a list of all toolbox directory names, enter the following command at the MATLAB prompt:

```
  dir([matlabroot '/toolbox'])
```

**Remarks**

The `verLessThan` function is available with MATLAB Version 7.4. If you are running a version of MATLAB earlier than 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 the proper 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.');
  end
```
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 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 number for MATLAB

As an alternative to the `version` function, select **About** from the **Help** menu in the MATLAB desktop.

### Syntax

```matlab
version
v = version
[v d] = version
version option
v = version('option')
```

### Description

`version` displays the MATLAB version number.

- `v = version` returns the MATLAB version number in `v`.
- `[v d] = version` also returns a string `d` containing the date of the version.

`version option` displays the following additional information about the version.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-date</td>
<td>Release date</td>
</tr>
<tr>
<td>-java</td>
<td>Java VM (JVM) version used by MATLAB</td>
</tr>
<tr>
<td>-release</td>
<td>Release number</td>
</tr>
</tbody>
</table>

`v = version('option')` returns additional information about the version. Valid string values for `option` are listed in the table above. You can only specify one output when using this syntax.

### Remarks

On Windows and UNIX platforms, MATLAB includes a JVM and uses that version. If you use the MATLAB Java interface and the Java classes you want to use require a different JVM than the version provided with MATLAB, it is possible to run MATLAB with a different
JVM. For details, see Solution 1-1812J on the MathWorks Support Web site.

On the Macintosh platform, MATLAB does not include a JVM, but uses whatever JVM is currently running on the machine.

**Examples**

```matlab
[v,d] = version
v =
   7.3.0.22078 (R2006b)

d =
   September 19, 2006
```

Run the following command in MATLAB R14 Service Pack 3:

```matlab
['Release R' version('-release') ', ' ... 
   version('-description')]
```

ans =
   Release R14, Service Pack 3

**See Also**

ver, whatsnew

*Help > Check for Updates* in the MATLAB desktop.
**Purpose**
Concatenate arrays vertically

**Syntax**
\[ \texttt{C = vertcat(A1, A2, ...)} \]

**Description**
\[ \texttt{C = vertcat(A1, A2, ...)} \]
vertically concatenates matrices \[ \texttt{A1}, \ \texttt{A2}, \ \text{and so on.} \]
All matrices in the argument list must have the same number of columns.

\texttt{vertcat} concatenates N-dimensional arrays along the first dimension.
The remaining dimensions must match.

MATLAB calls \[ \texttt{C = vertcat(A1, A2, ...)} \]
for the syntax \[ \texttt{C = [A1; A2; ...]} \] when any of \[ \texttt{A1}, \ \texttt{A2}, \ \text{etc.} \ ] is an object.

**Examples**
Create a 5-by-3 matrix, \( A \), and a 3-by-3 matrix, \( B \). Then vertically concatenate \( A \) and \( B \).

\[
A = \text{magic}(5); \quad \% \text{Create 5-by-3 matrix, } A \\
A(:, 4:5) = []
\]

\[
A = \\
\begin{array}{ccc}
17 & 24 & 1 \\
23 & 5 & 7 \\
4 & 6 & 13 \\
10 & 12 & 19 \\
11 & 18 & 25
\end{array}
\]

\[
B = \text{magic}(3) \times 100 \quad \% \text{Create 3-by-3 matrix, } B
\]

\[
B = \\
\begin{array}{ccc}
800 & 100 & 600 \\
300 & 500 & 700 \\
400 & 900 & 200
\end{array}
\]
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
vertcat (timeseries)

**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 = vertcat(tsc1,tsc2,...)

**Description**
tsc = 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
view

**Purpose**
Viewpoint specification

**Syntax**

```matlab
view(az,el)
view([x,y,z])
view(2)
view(3)
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(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°.
az = 180;
el = 90;
view(az, el);

See Also

viewmtx, hgtransform, rotate3d

“Controlling the Camera Viewpoint” on page 1-98 for related functions

Axes graphics object properties CameraPosition, CameraTarget,
CameraViewAngle, Projection

Defining the View for more information on viewing concepts and

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

```matlab
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
T = 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 = 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 1 1 1 1 1 0 0 1 1 0 1 0];

Transform the points in these vectors to the screen, then plot the object.

A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
Use a perspective transformation with a 25 degree viewing angle:

\[ A = \text{viewmtx}(-37.5,30,25); \]
\[ x4d = [.5 \, 0 \, -3 \, 1]'; \]
\[ x2d = A*x4d; \]
\[ x2d = x2d(1:2)/x2d(4) \quad \% \text{Normalize} \]
\[ x2d = \]

```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)
```

0.1777
-1.8858
See Also

view, hgtransform

“Controlling the Camera Viewpoint” on page 1-98 for related functions

Defining the View for more information on viewing concepts and techniques
volumebounds

Purpose
Coordinate and color limits for volume data

Syntax
lims = volumebounds(X,Y,Z,V)
lims = volumebounds(X,Y,Z,U,V,W)
lims = volumebounds(V), lims = volumebounds(U,V,W)

Description
lims = volumebounds(X,Y,Z,V) returns the x, y, z, and color limits of the current axes for scalar data. lims is returned as a vector:

\[\text{lims} = [\text{xmin} \ \text{xmax} \ \text{ymin} \ \text{ymax} \ \text{zmin} \ \text{zmax} \ \text{cmin} \ \text{cmax}]\]

You can pass this vector to the axis command.

lims = volumebounds(X,Y,Z,U,V,W) returns the x, y, and z limits of the current axes for vector data. lims is returned as a vector:

\[\text{lims} = [\text{xmin} \ \text{xmax} \ \text{ymin} \ \text{ymax} \ \text{zmin} \ \text{zmax}]\]

lims = volumebounds(V), lims = volumebounds(U,V,W) assumes X, Y, and Z are determined by the expression

\[\text{[X} \ \text{Y} \ \text{Z]} = \text{meshgrid}(1:n,1:m,1:p)\]

where \([m \ n \ p] = \text{size(V)}\).

Examples
This example uses volumebounds to set the axis and color limits for an isosurface generated by the flow function.

\[
\begin{align*}
[x \ y \ z \ v] &= \text{flow}; \\
p &= \text{patch(isosurface(x,y,z,v,-3))}; \\
isonormals(x,y,z,v,p) \\
daspect([1 \ 1 \ 1]) \\
isocolors(x,y,z,\text{flipdim(v,2)},p) \\
\text{shading interp} \\
\text{axis(volumebounds(x,y,z,v))} \\
\text{view(3)} \\
\text{camlight} \\
\text{lighting phong}
\end{align*}
\]
See Also

isosurface, streamslice

“Volume Visualization” on page 1-101 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.
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,'.');  
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.

```matlab
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.

```
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**

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 \( \text{numv} \)-by-\( n \) array of the \( \text{numv} \) 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. For an example, see “Tessellation and Interpolation of Scattered Data in Higher Dimensions” in the MATLAB Mathematics documentation.

**Examples**

**Example 1**

Let
\( x = \begin{bmatrix}
0.5 & 0 \\
0 & 0.5 \\
-0.5 & -0.5 \\
-0.2 & -0.1 \\
-0.1 & 0.1 \\
0.1 & -0.1 \\
0.1 & 0.1 \\
\end{bmatrix} \)

then

\[ [V,C] = \text{voronoin}(x) \]

\[ V = \begin{bmatrix}
\text{Inf} & \text{Inf} \\
0.3833 & 0.3833 \\
0.7000 & -1.6500 \\
0.2875 & 0.0000 \\
-0.0000 & 0.2875 \\
-0.0000 & -0.0000 \\
-0.0500 & -0.5250 \\
-0.0500 & -0.0500 \\
-1.7500 & 0.7500 \\
-1.4500 & 0.6500 \\
\end{bmatrix} \]

\[ C = \begin{bmatrix}
[1x4 \text{ double}] \\
[1x5 \text{ double}] \\
[1x4 \text{ double}] \\
[1x4 \text{ double}] \\
[1x4 \text{ double}] \\
[1x4 \text{ double}] \\
[1x5 \text{ double}] \\
[1x4 \text{ double}] \\
\end{bmatrix} \]

Use a for loop to see the contents of the cell array \( C \).

\[
\text{for } i=1:\text{length}(C), \text{ disp}(C\{i\}), \text{ end}
\]

\[
4 \quad 2 \quad 1 \quad 3
\]
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`.

\[
[V,C] = voronoin(X,{'Qbb','Qz'})
\]

\[
V =
\begin{array}{cc}
\text{Inf} & \text{Inf} \\
0 & 0
\end{array}
\]

\[
C =
\begin{array}{c}
[1x2 \text{ double}] \\
[1x2 \text{ double}]
\end{array}
\]
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

wait

**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 waitbar

Syntax
h = waitbar(x,'message')
waitbar(x,'message','CreateCancelButton','button_callback')
waitbar(...,property_name,property_value,...)
waitbar(x)
waitbar(x,h)
waitbar(x,h,'updated message')

Description
A waitbar shows what percentage of a calculation is complete, as the
calculation proceeds.

h = waitbar(x,'message') displays a waitbar of fractional length x.
The waitbar figure is modal. Its handle is returned in h. The argument x
must be between 0 and 1.

Note A modal figure prevents the user from interacting with other
windows before responding. For more information, see WindowStyle in
the MATLAB Figure Properties.

waitbar(x,'message','CreateCancelButton','button_callback')
specifying CreateCancelButton 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.

waitbar(...,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.

waitbar(x,h) extends the length of the bar in the waitbar 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.

**Example**

waitbar is typically used inside a for loop that performs a lengthy computation. For example,

```matlab
h = waitbar(0,'Please wait...');
for i=1:100, % computation here%
    waitbar(i/100)
end
close(h)
```

**See Also**

“Predefined Dialog Boxes” on page 1-103 for related functions
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.

`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).

`waitfor` can block nested execution streams. For example, a callback invoked during a `waitfor` statement can itself invoke `waitfor`.

See Also

uiresume, uiwait

“Developing User Interfaces” on page 1-104 for related functions
Purpose
Wait for key press or mouse-button click

Syntax
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.

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

“Developing User Interfaces” on page 1-104 for related functions
Purpose
Open warning dialog box

Syntax
h = warndlg
h = warndlg(warningstring)
h = warndlg(warningstring,dlgname)

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

    warndlg(sprintf('Message line 1 \n Message line 2'))

• Cell arrays of strings

    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. 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

```matlab
warndlg('Pressing OK will clear memory','!! Warning !!')
```
displays this dialog box:

![Warning Dialog Box]

See Also
dialog, errordlg, helpdlg, inputdlg, listdlg, msgbox, questdlg, figure, uiwait, uiresume, warning

“Predefined Dialog Boxes” on page 1-103 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 the text 'message' like the disp function, except that with warning, message display can be suppressed.

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 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 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 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 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 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 actionNotTaken 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:

```matlab
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:

```matlab
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:

```matlab
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:

```matlab
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(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 CLim 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'.

2-3570
For a discussion of parametric surfaces and related color properties, see surf.

**See Also**

axes, axis, caxis, meshz, ribbon, 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**

wavread
**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; F_s1 = F_s;
load gong;
wavplay(y1,F_s1,'sync') % The chirp signal finishes before the gong signal begins playing.
wavplay(y,F_s) % gong signal begins playing.
```

See Also

wavrecord
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
\[
y = \text{wavread}(\text{filename})
\]
\[
[y, Fs, nbits] = \text{wavread}(\text{filename})
\]
\[
[\ldots] = \text{wavread}(\text{filename}, N)
\]
\[
[\ldots] = \text{wavread}(\text{filename},[N1 N2])
\]
\[
y = \text{wavread}(\text{filename}, \text{fmt})
\]
\[
siz = \text{wavread}(\text{filename},'\text{size}')
\]
\[
[y, fs, nbits, opts] = \text{wavread}(\ldots)
\]

Description
\( y = \text{wavread}(\text{filename}) \) 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.

\( [y, Fs, nbits] = \text{wavread}(\text{filename}) \) returns the sample rate (Fs) in Hertz and the number of bits per sample (nbits) used to encode the data in the file.

\( [\ldots] = \text{wavread}(\text{filename}, N) \) returns only the first \( N \) samples from each channel in the file.

\( [\ldots] = \text{wavread}(\text{filename},[N1 N2]) \) returns only samples \( N1 \) through \( N2 \) from each channel in the file.

\( y = \text{wavread}(\text{filename}, \text{fmt}) \) specifies the data type format of \( y \) used to represent samples read from the file. \( \text{fmt} \) can be either of the following values.
Value | Description
---|---
'double' | $y$ contains double-precision normalized samples. This is the default value, if $fmt$ is omitted.
'native' | $y$ contains samples in the native data type found in the file. Interpretation of $fmt$ is case-insensitive, and partial matching is supported.

$siz = wavread(filename,'size')$ returns the size of the audio data contained in $filename$ in place of the actual audio data, returning the vector $siz = [\text{samples channels}]$.

$[y, fs, nbits, opts] = wavread(\ldots)$ 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>$0 \leq y \leq 255$</td>
</tr>
<tr>
<td>16</td>
<td>int16 (signed integer)</td>
<td>$-32768 \leq y \leq +32767$</td>
</tr>
<tr>
<td>24</td>
<td>int32 (signed integer)</td>
<td>$-2^{23} \leq y \leq 2^{23} - 1$</td>
</tr>
<tr>
<td>32</td>
<td>single (floating point)</td>
<td>$-1.0 \leq y \leq +1.0$</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 may achieve +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, wavwrite
**Purpose**
Record sound using PC-based audio input device

**Syntax**

\[
y = \text{wavrecord}(n,Fs)\\
y = \text{wavrecord}(\ldots,ch)\\
y = \text{wavrecord}(\ldots,\text{'dtype'})
\]

**Description**

\[y = \text{wavrecord}(n,Fs)\] 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.

\[y = \text{wavrecord}(\ldots,ch)\] 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.

\[y = \text{wavrecord}(\ldots,\text{'dtype'})\] uses the data type specified by the string 'dtype' to record the sound. The string 'dtype' can be one of the following:

- 'double' (default value), 16 bits/sample
- 'single', 16 bits/sample
- 'int16', 16 bits/sample
- 'uint8', 8 bits/sample

**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.

**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        wavplay
wavwrite

**Purpose**
Write Microsoft WAVE (.wav) sound file

**Syntax**
wavwrite(y,filename)
wavwrite(y,Fs,filename)
wavwrite(y,Fs,N,filename)

**Description**
wavwrite writes data to 8-, 16-, 24-, and 32-bit .wav files.

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. Amplitude values outside the range [-1, +1] are clipped prior to writing.

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. Amplitude values outside the range [-1, +1] are clipped prior to writing.

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. For N < 32, amplitude values outside the range [-1, +1] are clipped.

**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, wavread
**Purpose**
Open Web site or file in Web browser or Help browser

**Syntax**
- `web url`
- `web url -new`
- `web url -notoolbar`
- `web url -noaddressbox`
- `web url -helpbrowser`
- `web url -browser`
- `web(...)`
  ```
  stat = web('url', '-browser')
  [stat, h1] = web
  [stat, h1, url] = web
  ```

**Description**
`web` opens an empty MATLAB “Web Browser”. The MATLAB Web browser includes an address field where you can enter a URL, for example, to a Web site or file, a toolbar with common browser buttons, and a MATLAB desktop menu.

`web url` displays the specified URL, `url`, in the MATLAB Web browser. If any MATLAB Web browsers are already open, it displays the page in the browser that last had focus. Files up to 1.5MB in size display in the MATLAB Web browser, while larger files instead display in the default Web browser for your system. If `url` is located in the directory returned when you run `docroot` (an unsupported utility), the URL displays in the MATLAB Help browser instead of the MATLAB Web browser.

`web url -new` displays the specified URL, `url`, in a new MATLAB Web browser.

`web url -notoolbar` displays the specified URL, `url`, in a MATLAB Web browser that does not include the toolbar and address field. If any MATLAB Web browsers are already open, also use the `-new` option; otherwise `url` displays in the browser that last had focus, regardless of its toolbar status.

`web url -noaddressbox` displays the specified URL, `url`, in a MATLAB Web browser that does not include the address field. If any MATLAB Web browsers are already open, also use the `-new` option; otherwise `url`
web displays in the browser that last had focus, regardless of its address field status.

`web url -helpbrowser` displays the specified URL, `url`, in the MATLAB Help browser.

`web url -browser` displays the default Web browser for your system and loads the file or Web site specified by the URL `url` in it. Generally, `url` specifies a local file or a Web site on the Internet. The URL can be in any form that the browser supports. On Windows and Macintosh, the default Web browser is determined by the operating system. On UNIX, the Web browser used is specified via `docopt` in the `doccmd` string.

`web(...)` is the functional form of `web`.

`stat = web('url', '-browser')` runs `web` and returns the status of `web` to the variable `stat`.

<table>
<thead>
<tr>
<th>Value of stat</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Browser was found and launched.</td>
</tr>
<tr>
<td>1</td>
<td>Browser was not found.</td>
</tr>
<tr>
<td>2</td>
<td>Browser was found but could not be launched.</td>
</tr>
</tbody>
</table>

`[stat, h1] = web` returns the status of `web` to the variable `stat`, and returns a handle to the Java class, `h1`, for the last active browser.

`[stat, h1, url] = web` returns the status of `web` to the variable `stat`, returns a handle to the Java class `h1`, for the last active browser, and returns its current URL to `url`.

**Examples**

Run

```
web http://www.mathtools.net
```

and MATLAB displays

web file:///disk/dir1/dir2/foo.html opens the file foo.html in the MATLAB Web browser.

web(['file:///' which('foo.html')]) opens foo.html if the file is on the MATLAB path or in the current directory.

web('text://<html><h1>Hello World</h1></html>') displays the HTML-formatted text Hello World.

web ('http://www.mathworks.com', '-new', '-notoolbar') loads the MathWorks Web site home page into a new MATLAB Web browser that does not include a toolbar or address field.

web file:///disk/dir1/foo.html -helpbrowser opens the file foo.html in the MATLAB Help browser.
web file:///disk/dir1/foo.html -browser opens the file foo.html in the system Web browser.

web mailto:email_address uses your system browser's default e-mail application to send a message to email_address.

web http://www.mathtools.net -browser opens a browser to mathtools.net. Then [stat,h1,url]=web returns

    stat =
        0

    h1 =
       com.mathworks.mde.webbrowser.WebBrowser[,0,0,591x140,
          layout=java.awt.BorderLayout,alignmentX=null,alignmentY=null,
          border=,flags=9,maximumSize=,minimumSize=,preferredSize=]

    url =
       http://www.mathtools.net/

Run methods(h1) to view allowable methods for the class. As an example, you can use the method setCurrentLocation to change the URL displayed in h1, as in

    setCurrentLocation(h1,'http://www.mathworks.com')

See Also
doc, docopt, helpbrowser, matlabcolon

"Web Browser" in the MATLAB Desktop Tools and Development Environment documentation
**Purpose**
Day of week

**Syntax**

\[
\begin{align*}
\{N, S\} &= \text{weekday}(D) \\
\{N, S\} &= \text{weekday}(D, \text{form}) \\
\{N, S\} &= \text{weekday}(D, \text{locale}) \\
\{N, S\} &= \text{weekday}(D, \text{form}, \text{locale})
\end{align*}
\]

**Description**

\[
\{N, S\} = \text{weekday}(D) \text{ returns the day of the week in numeric (N) and string (S) form for a given serial date number or date string D. Input argument D can represent more than one date in an array of serial date numbers or a cell array of date strings.}
\]

\[
\{N, S\} = \text{weekday}(D, \text{form}) \text{ returns the day of the week in numeric (N) and string (S) form, where the content of S depends on the form argument. If form is 'long', then S contains the full name of the weekday (e.g., Tuesday). If form is 'short', then S contains an abbreviated name (e.g., Tues) from this table.}
\]

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>

\[
\{N, S\} = \text{weekday}(D, \text{locale}) \text{ returns the day of the week in numeric (N) and string (S) form, where the format of the output depends on the locale argument. If locale is 'local', then weekday uses local format for its output. If locale is 'en_US', then weekday uses US English.}
\]
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”. To open it, select **Current Directory** from the **Desktop** menu in the MATLAB desktop.

**Syntax**

```matlab
what
what dirname
what class
s = what('dirname')
```

**Description**

`what` lists the M, MAT, MEX, MDL, and P-files and the class directories that reside in the current working directory.

`what dirname` lists the files in directory `dirname` on the MATLAB search path. It is not necessary to enter the full pathname of the directory. The last component, or last two components, is sufficient.

`what class` lists the files in method directory, `@class`. For example, `what cfit` lists the MATLAB files in `toolbox/curvefit/curvefit/@cfit`.

`s = what('dirname')` returns the results in a structure array with these fields.

<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th><strong>Description</strong></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>
<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 names</td>
</tr>
</tbody>
</table>
Examples

List the files in toolbox/matlab/audiovideo:

```plaintext
what audiovideo

M-files in directory matlabroot\toolbox\matlab\audiovideo

Contents       aviinfo           render_uimgraudiotoolbar
audiodevinfo   aviread           sound
audioplayerreg lin2mu            soundsc
audiorecorderreg mmcompinfo      wavfinfo
audiouniquenamen mmfileinfo      wavplay
aufinfo         movie2avi         wavread
auread          mu2lin            wavrecord
auwrite         prefspanel        wavwrite
avifinfo        render_fullaudiotoolbar

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    audiorecorder    avifile    mmreader

Obtain a structure array containing the MATLAB filenames in toolbox/matlab/general:

```plaintext
s = what('general')
s =
    path: 'matlabroot:\toolbox\matlab\general'
    m: {87x1 cell}
    mat: {0x1 cell}
mex: {2x1 cell}
mdl: {0x1 cell}
p: {'callgraphviz.p'}
classes: {0x1 cell}

See Also  dir, exist, lookfor, mfilename, path, which, who
whatsnew

**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, use the “Current Directory Browser”.

**Syntax**

```plaintext
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`.

`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.

2-3591
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. 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.

    which pinv
    matlabroot\toolbox\matlab\matfun\pinv.m
To find the fopen function used on MATLAB serial class objects

    which serial/fopen
    matlabroot\toolbox\matlab\iofun\@serial\fopen.m  % serial method

To find the setTitle method used on objects of the Java Frame class, the class must first be loaded into MATLAB. The class is loaded when you create an instance of the class:

    frameObj = java.awt.Frame;

    which setTitle
    java.awt.Frame.setTitle  % Frame 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:

    s = which('private/stradd',''-all'');
    whos s
        Name      Size     Bytes  Class
    s         3x1      562  cell array
    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` with the value of the loop counter incremented by 1.

`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 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 2-3596, 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.

while \((b \neq 0) && (a/b > 18.5)\)
if exist('myfun.m') && (myfun(x) >= y)
if iscell(A) && 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

A = [];
while all(A), do_something, end
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,

```matlab
A =
    1   0
    2   3

B =
    1   1
    3   4
```

<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>
### Table

<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
whitebg

Purpose
Change axes background color

Syntax
whitebg
whitebg(fig)
whitebg(ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig)

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.
Examples

Set the background color to blue-gray.

    whitebg([0 .5 .6])

Set the background color to blue.

    whitebg('blue')

See Also

ColorSpec, colordef
The figure graphics object property InvertHardCopy
“Color Operations” on page 1-97 for related functions
who, whos

**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 apply 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, MATLAB 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>whos('global')</td>
</tr>
<tr>
<td>'-file', filename</td>
<td>List variables in the specified MAT-file. Use the full path for filename.</td>
<td>whos('-file', 'mydata')</td>
</tr>
<tr>
<td>'-regexp', exprlist</td>
<td>List variables that match any of the regular expressions in exprlist.</td>
<td>whos('-regexp', '[AB].', '\w\d')</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, whos

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td>True if the variable is global; otherwise false</td>
</tr>
<tr>
<td>sparse</td>
<td>True if the variable is sparse; otherwise false</td>
</tr>
<tr>
<td>complex</td>
<td>True if the variable is complex; otherwise false</td>
</tr>
<tr>
<td>nesting</td>
<td>Structure having the following fields:</td>
</tr>
<tr>
<td></td>
<td>• function — Name of the nested or outer function that defines the variable</td>
</tr>
<tr>
<td></td>
<td>• level — Nesting level of that function</td>
</tr>
<tr>
<td>persistent</td>
<td>True if the variable is persistent; otherwise false</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`:

```matlab
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:

```matlab
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:

```matlab
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:
```

```matlab
show_attributes```
<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 = datenow;

[ 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1x3</td>
<td>378</td>
<td>cell</td>
</tr>
<tr>
<td>day</td>
<td>1x1</td>
<td>64</td>
<td>cell</td>
</tr>
<tr>
<td>mon</td>
<td>1x1</td>
<td>66</td>
<td>cell</td>
</tr>
<tr>
<td>year</td>
<td>1x1</td>
<td>68</td>
<td>cell</td>
</tr>
<tr>
<td>ans</td>
<td>0x0</td>
<td>0</td>
<td>(unassigned)</td>
</tr>
<tr>
<td>date_time</td>
<td>1x20</td>
<td>40</td>
<td>char</td>
</tr>
<tr>
<td>pos</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>str</td>
<td>1x3</td>
<td>378</td>
<td>cell</td>
</tr>
</tbody>
</table>

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`
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 0 1 1 1
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. 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 Microsoft Word:

```plaintext
    winopen('thesis.doc')
```

Open `myresults.html` in your system’s default Web browser:

```plaintext
    winopen('D:/myfiles/myresults.html')
```

**See Also**  
dos, open, web
**Purpose**

Item from Microsoft 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` in 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 MATLAB 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 COM control `mwsampctrl1.2`:

```matlab
winqueryreg 'HKEY_CLASSES_ROOT' 'mwsamp.mwsampctrl1.2\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:

[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**

\[
\begin{align*}
M &= \text{wk1read(filename)} \\
M &= \text{wk1read(filename,r,c)} \\
M &= \text{wk1read(filename,r,c,range)}
\end{align*}
\]

**Description**

- \(M = \text{wk1read(filename)}\) reads a Lotus1-2-3 WK1 spreadsheet file into the matrix \(M\). The filename input is a string enclosed in single quotes.
- \(M = \text{wk1read(filename,r,c)}\) 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.
- \(M = \text{wk1read(filename,r,c,range)}\) 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
    
    \[
    [\text{upper_left_row} \ \text{upper_left_col} \ \text{lower_right_row} \ \text{lower_right_col}]
    \]
  - 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:

\[
\begin{align*}
A &= [1:8 \ 11:18 \ 21:28 \ 31:38 \ 41:48 \ 51:58 \ 61:68 \ 71:78] \\
A &=
\end{align*}
\]
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:

\[
A = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
11 & 12 & 13 & 14 & 15 \\
21 & 22 & 23 & 24 & 25 \\
31 & 32 & 33 & 34 & 35 \\
\end{bmatrix}
\]

wk1write('matA.wk1', A, 2, 3)

M = wk1read('matA.wk1')
M =
See Also  wk1read, dlmwrite, dlmread, csvwrite, csvread
**workspace**

**Purpose**
Open Workspace browser to manage workspace

**GUI Alternatives**
As an alternative to the `workspace` function, select **Desktop > Workspace** in the MATLAB desktop.

**Syntax**
`workspace`

**Description**
`workspace` displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the MATLAB workspace. It provides a graphical representation of the `whos` display, and allows you to perform the equivalent of the `clear`, `load`, `open`, and `save` functions.

The Workspace browser also displays and automatically updates statistical calculations for each variable that you can choose to show or hide.

![Workspace Browser](image)

You can edit the 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 types, double-click the variable in the Workspace browser. The variable displays in the Array Editor, where you can view the full contents and edit it.
See Also

who
**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 1-86 for related functions

Adding Axis Labels to Graphs for more information about labeling axes
**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**

```matlab
xlim
xlim([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])
```

![3D graph showing an example of using xlim and ylim](image)
**xlim, ylim, zlim**

**See Also**

- `axis`
  - The axes properties `XLim, YLim, ZLim`
  - “Setting the Aspect Ratio and Axis Limits” on page 1-99 for related functions
  - Understanding Axes Aspect Ratio for more information on how axis limits affect the axes
Purpose
Determine whether file contains Microsoft Excel (.xls) spreadsheet

Syntax
\[
\text{typ} = \text{xlsfinfo}(\text{filename}) \\
[\text{typ}, \text{desc}] = \text{xlsfinfo}(\text{filename}) \\
[\text{typ}, \text{desc}, \text{fmt}] = \text{xlsfinfo}(\text{filename}) \\
\text{xlsfinfo} \text{ filename}
\]

Description
\(\text{typ} = \text{xlsfinfo}(\text{filename})\) returns the string 'Microsoft Excel Spreadsheet' if the file specified by \text{filename} is an XLS file that can be read by the MATLAB \text{xlsread} function. Otherwise, \text{typ} is the empty string, (''). The \text{filename} input is a string enclosed in single quotes.

\([\text{typ}, \text{desc}] = \text{xlsfinfo}(\text{filename})\) returns in \text{desc} a cell array of strings containing the names of each spreadsheet in the file. If a spreadsheet is unreadable, the cell in \text{desc} that represents that spreadsheet contains an error message.

\([\text{typ}, \text{desc}, \text{fmt}] = \text{xlsfinfo}(\text{filename})\) returns in the \text{fmt} output a string containing the actual format of the file as obtained from the Excel COM server. On UNIX systems, or on Windows when the COM server is not available, \text{fmt} is returned as an empty string, ('').

Note
In the case where an Excel COM server cannot be started, functionality is limited in that some Excel files might not be readable.

Get information about an .xls file:

\[
[\text{typ}, \text{desc}, \text{fmt}] = \text{xlsfinfo}('\text{myaccount.xls}')
\]

\text{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 =
        xlXMLSpreadsheet

See Also    xlsread, xlswrite

2-3624
Purpose

Read Microsoft Excel spreadsheet file (.xls)

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 quotes.

`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, one or more header lines appearing at the top of a spreadsheet are ignored by `xlsread`. Any inner rows or columns in which some or all cells contain nonnumeric data are not ignored. The nonnumeric cells are instead assigned a value of NaN.

The full functionality of `xlsread` depends on the ability to start Excel as a COM server from MATLAB. If your system does not have this capability, the `xlsread` syntax that passes the 'basic' keyword is recommended. As long as the COM server is available, you can use `xlsread` on Excel files having formats other than XLS (for example, HTML).
xlsxread

**Note** xlsxread on UNIX is being grandfathered. If the Excel COM server is not available, xlsxread reads only strictly XLS files. It cannot read Excel files saved in HTML or other formats.

`num = xlsxread(filename, -1)` opens the file `filename` in an Excel window, enabling you to interactively select the worksheet to be 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 2-3629 below.)

`num = xlsxread(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 `xlsxinfo`.

`num = xlsxread(filename, 'range')` reads data from a specific rectangular region of the default worksheet (Sheet1). 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 notation. (See help in Excel for more information on this notation.) (Also, see “COM Server Requirements” on page 2-3629 below.)

`num = xlsxread(filename, sheet, 'range')` reads data from a specific rectangular region (`range`) of the worksheet specified by `sheet`. See the previous two syntax formats for further explanation of the sheet and range inputs. (See “COM Server Requirements” on page 2-3629 below.)

`num = xlsxread(filename, sheet, 'range', 'basic')` imports data from the spreadsheet in basic import mode. This is the mode used on UNIX platforms as well as on Windows when Excel is not available as a COM server. In this mode, xlsxread does not use Excel as a COM server,
and this limits import ability. Without Excel as a COM server, range is ignored and, consequently, the whole active range of a sheet is imported. (You can set range to the empty string (''). Also, in basic mode, sheet is case-sensitive and must be a quoted string.

\(\text{num} = \text{xlsread(filename, \ldots, functionhandle)}\) 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 2-3629 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 Excel 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.

\([	ext{num}, \text{txt}] = \text{xlsread(filename, \ldots)}\) 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.

If txt includes data that was previously written to the file using xlswrite, and the range specified for that xlswrite operation caused undefined data ('#N/A') to be written to the worksheet, then cells containing that undefined data are represented in the txt output as 'ActiveX VT_ERROR: '.

\([	ext{num}, \text{txt}, \text{raw}] = \text{xlsread(filename, \ldots)}\) 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 2-3629 below.)

\([	ext{num}, \text{txt}, \text{raw}, \text{X}] = \text{xlsread(filename, \ldots, functionhandle)}\) 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 2-3629 below.)

 xlsread filename sheet range basic is 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

Handling Excel Date Values
MATLAB imports date fields from Excel files in the format in which they were stored in the Excel file. If stored in string or date format, xlsread returns the date as a string. If stored in a numeric format, xlsread returns a numeric date.

Both Excel and MATLAB represent numeric dates as a number of serial days elapsed from a specific reference date. However, Excel uses January 1, 1900 as the reference date while MATLAB uses January 0, 0000. Due to this difference in the way Excel and MATLAB compute numeric date values, any numeric date imported from Excel into MATLAB must first be converted before being used in the MATLAB application.

You can do this conversion after the xlsread completes, as shown below:

```matlab
excelDates = xlsread(filename)
matlabDates = datenum('30-Dec-1899') + excelDates
datestr(matlabDates,2)
```

You can also do this as part of the xlsread operation by writing a conversion routine that acts directly on the Excel COM Range object, and then passing a function handle for your routine as an input to xlsread. The description above for the following syntax, along with Examples 5 and 6, explain how to do this:

```matlab
[num, txt, raw, X] = xlsread(filename, ..., functionhandle)
```
**COM Server Requirements**

The following six syntax formats are supported only on computer systems capable of starting Excel as a COM server from MATLAB. They are not supported in basic mode.

```matlab
num = xlsread(filename, -1)
num = xlsread(filename, 'range')
num = xlsread(filename, sheet, 'range')
[num, txt, raw] = xlsread(filename, ...)
num = xlsread(filename, ..., functionhandle)
[num, txt, raw, opt] = xlsread(filename, ..., functionhandle)
```

**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:

```matlab
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
```
xlsread puts a NaN in place of the text data in the result:

```matlab
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':

```matlab
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. xlsread ignores any leading row or column of text in the numeric result.
ndata = xlsread('tempdata.xls', 'Temperatures')

ndata =
  12   98
  13   99
  14   97

To import both the numeric data and the text data, specify two return values for xlsread:

[ndata, headertext] = xlsread('tempdata.xls', 'Temperatures')

ndata =
  12   98
  13   99
  14   97

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. These modifications are performed by a user-written function, setMinMax, that you pass as a function handle in the call to xlsread. 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. Note that the values returned 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 will need to pass this function in the call to `xlsread` which will then execute the function on the data it has read before returning it to you.

When `xlsread` calls your function, it passes a range interface from Excel 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{1};
    if v > maxval || v < minval
      for k = 1:DataRange.Count
        if v > maxval
          DataRange.Value{1} = maxval;
      end
    end
  end
```

2-3632
Now call `xlsread`, passing a function handle for the `setMinMax` function as the final argument. Note the changes from the values returned from the last call to `xlsread`:

```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 reports which elements of the spreadsheet matrix have been altered. 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};
```
xlsread

if v > maxval || v < minval
    if v > maxval
        DataRange.Value{k} = maxval;
    else
        DataRange.Value{k} = 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. Call xlsread again, and you will see just where the returned matrix has been modified:

[arr txt raw idx] = xlsread('testsheet.xls', ...
    '', '', '', @setMinMax);

idx
idx =
    3  5  8  11  13  15  16  17  22  24  25  28  30
arr
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

See Also    xlswrite, xlsinfo, wk1read, textread, function_handle

2-3634
**Purpose**

Write Microsoft Excel spreadsheet file (.xls)

**Syntax**

```
xlswrite(filename, M)
xlswrite(filename, M, sheet)
xlswrite(filename, M, 'range')
xlswrite(filename, M, sheet, 'range')
status = xlswrite(filename, ...)
[status, message] = xlswrite(filename, ...)
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 quotes. The input matrix `M` is an `m`-by-`n` numeric, character, or cell array, where `m < 65536` and `n < 256`. The matrix data is written to the first worksheet in the file, starting at cell A1.

`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.

If `sheet` does not exist, a new sheet is added at the end of the worksheet collection. If `sheet` is an index larger than the number of worksheets, empty sheets are appended until the number of worksheets in the workbook equals `sheet`. In either case, MATLAB 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 one of the following quoted string formats:

- A cell designation, such as 'D2', to indicate the upper left corner of the region to receive the matrix data.
- Two cell designations separated by a colon, such as 'D2:H4', to indicate two opposing corners of the region to receive the matrix data. 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 Excel A1 notation. (See help in Excel for more information on this notation.)

The size defined by range should fit the size of M or contain only the first cell, (e.g., 'A2'). If range is larger than the size of M, Excel fills the remainder of the region with #N/A. If range is smaller than the size of M, only the submatrix that fits into range is written to the file specified by filename.

`xlswrite(filename, M, sheet, 'range')` writes matrix M to a rectangular region specified by range in worksheet sheet of the file filename. 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 completed successfully, status is equal to logical 1 (true). Otherwise, status is logical 0 (false). Unless you specify an output for xlswrite, no status is displayed 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 quotation marks are required around the string (for example, 'Income 2002').
The above functionality depends upon having Microsoft Excel as a COM server. In absence of Excel, matrix M is written as a text file in Comma-Separated Value (CSV) format. In this mode, the sheet and range arguments are ignored.

Examples

Example 1 — Writing Numeric Data to the Default Worksheet

Write a 7-element vector to Microsoft Excel file testdata.xls. By default, the data is written to cells A1 through G1 in the first worksheet in the file:

```matlab
xlswrite('testdata', [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:

```matlab
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. The 4-by-2 matrix will be written to the rectangular region that starts at cell E1 in its upper left corner:

```matlab
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:

<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>
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, MATLAB appends a new sheet to the workbook, calling it by the name you supplied in the sheets input argument, 'NewTemp'. MATLAB 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 using the command indicated in the message above:

```matlab
warning off MATLAB:xlswrite:AddSheet
```

Now try the command again, this time creating another new worksheet, NewTemp2. Although the message is not displayed this time, you can still retrieve it and its identifier from the second output argument, m:

```matlab
[stat msg] = xlswrite('tempdata.xls', d, 'NewTemp2', 'E1');
msg
msg =
    message: 'Added specified worksheet.'
    identifier: 'MATLAB:xlswrite:AddSheet'
```

See Also

xlsread, xlsinfo, wk1read, textread
**Purpose**
Parse XML document and return Document Object Model node

**Syntax**

```matlab
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
```

% End script
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.');
end

% ---- Subfunction PARSECHILDNODES ----
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.nodeName), ... 
    'Attributes', parseAttributes(theNode), ... 
    'Data', '', ... 
    'Children', parseChildNodes(theNode));

if any(strcmp(methods(theNode), 'getData'))
    nodeStruct.Data = char(theNode.getData);
else
    nodeStruct.Data = '';
end
end

% ----- Subfunction PARSEATTRIBUTES ----- 
function attributes = parseAttributes(theNode) 
% Create attributes structure.

attributes = [];
if theNode.hasAttributes 
    theAttributes = theNode.getAttributes;

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
end

See Also       xmlwrite, xslt
xmlwrite

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

2-3644
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 \( C(i,j,\ldots) \) is logical true (1) if \( A(i,j,\ldots) \) or \( B(i,j,\ldots) \), but not both, is nonzero.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<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 \( A = [0 \ 0 \ \pi \ \text{eps}] \) and \( B = [0 \ -2.4 \ 0 \ 1] \), then

\[
\begin{align*}
C &= \text{xor}(A,B) \\
C &= \left[
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
\end{array}
\right]
\end{align*}
\]

To see where either \( A \) or \( B \) has a nonzero element and the other matrix does not,

\[
\text{spy(xor(A,B))}
\]

See Also

all, any, find, Elementwise Logical Operators, Short-Circuit Logical Operators
**Purpose**
Transform XML document using XSLT engine

**Syntax**
```matlab
result = xslt(source, style, dest)
[result,style] = xslt(...)  
xslt(...,'-web')
```

**Description**
`result = xslt(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:

- `source` is the filename or URL of the source XML file. `source` can also specify a DOM node.
- `style` is the filename or URL of an XSL stylesheet.
- `dest` is the filename or URL of the desired output document. If `dest` is absent or empty, the function uses a temporary filename. If `dest` is `-tostring`, the function returns the output document as a MATLAB string.

`[result,style] = xslt(...)` returns a processed stylesheet appropriate for passing to subsequent XSLT calls as `style`. This prevents costly repeated processing of the stylesheet.

`xslt(...,'-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.

```matlab
xslt info.xml info.xsl info.html -web
```
See Also  xmlread, xmlwrite
### Purpose
Create array of all zeros

### Syntax
- `B = zeros(n)`
- `B = zeros(m,n)`
- `B = zeros([m n])`
- `B = zeros(m,n,p,...)`
- `B = zeros([m n p ...])`
- `B = zeros(size(A))`
- `zeros(m, n,...,classname)`
- `zeros([m,n,...],classname)`

### Description
- `B = zeros(n)` returns an n-by-n matrix of zeros. An error message appears if n is not a scalar.
- `B = zeros(m,n)` or `B = zeros([m n])` returns an m-by-n matrix of zeros.
- `B = zeros(m,n,p,...)` or `B = 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 = zeros(size(A))` returns an array the same size as A consisting of all zeros.
- `zeros(m, n,...,classname)` or `zeros([m,n,...],classname)` is an m-by-n-by-... array of zeros of data type `classname`. `classname` is a string specifying the data type of the output. `classname` can have the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

### Example
```
x = zeros(2,3,'int8');
```

### 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. The `.zip` extension is appended to `zipfile` if omitted.

`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 pathnames. 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 filename or partial pathname.

`zip(zipfile, files, rootdir)` allows the path for `files` to be specified relative to `rootdir` rather than the current directory.

`entrynames = zip(...)` returns a string cell array of the relative path entry names contained in `zipfile`.

**Examples**

**Zip a File**

Create a zip file of the file `guide.viewlet`, which is in the demos directory of MATLAB. It saves the zip file in `d:/mymfiles/viewlet.zip`.

```matlab
file = fullfile(matlabroot,'demos','guide.viewlet');
zip('d:/mymfiles/viewlet.zip',file)
```
Run `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 `zip` for all `.m` and `.mat` files in the current directory to the file `backup.zip`:

```matlab
zip('backup', {'*.m', '*.mat'});
```

**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.

```matlab
zip('./mymfiles', 'D:/mymfiles')
```

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.

```matlab
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 Alternatives**

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 “Zooming in 2-D and 3-D” in the MATLAB Graphics documentation.

**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 $1/\text{factor}$.

`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. This read-only property 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.

*ButtonDownFilter* `<function_handle>`

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 callbacks), as follows:

```matlab
function [res] = myfunction(obj,event_obj)
    % OBJ handle to the object that has been clicked on.
    % EVENT_OBJ handle to event object (empty in this release).
    % RES 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>`

Set this callback to listen to when a zoom operation starts. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

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

The event object has the following read-only property:

| Axes                     | The handle of the axes that is being zoomed |
ActionPostCallback <function_handle>

Set this callback to listen to when a zoom operation finishes. The input function handle should reference a function with two implicit arguments (similar to handle callbacks), as follows:

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

UIContextMenu <handle>

Specifies a custom context menu to be displayed during a right-click action. This property is ignored if the 'RightClickZoomOut' property has been set to 'on'.

```matlab
flags = isAllowAxesZoom(h, 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.

```matlab
setAllowAxesZoom(h, axes, flag)
```

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.

```matlab
info = getAxesZoomMotion(h, axes)
```

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'.

```matlab
setAxesZoomMotion(h, axes, style)
```
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**

Simple zoom:

```matlab
plot(1:10);
zoom on
% zoom in on the plot
```

**Example 2**

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**

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**

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**

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);
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.');
end

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

**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.

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 1-99 for related functions
& 2-48 2-50
' 2-36
* 2-36
+ 2-36
- 2-36
/ 2-36
: 2-57
< 2-46
> 2-46
@ 2-1296
\ 2-36
^ 2-36
| 2-48 2-50
~ 2-48 2-50
&& 2-50
== 2-46
}) 2-56
|| 2-50
~= 2-46
1-norm 2-2207 2-2600
2-norm (estimate of) 2-2209

A

abs 2-59
absolute accuracy
   BVP 2-420
   DDE 2-806
   ODE 2-2254
absolute value 2-59
Accelerator
   Uimenu property 2-3395
accumarray 2-60
accuracy
   of linear equation solution 2-607
   of matrix inversion 2-607
acos 2-66
acosd 2-68
acosh 2-69
acot 2-71
acotd 2-73
acoth 2-74
acsc 2-76
acscd 2-78
acsch 2-79
activelegend 1-86 2-2429
actxcontrol 2-81
actxcontrollist 2-88
actxcontrolselect 2-89
actxserver 2-93
Adams-Bashforth-Moulton ODE solver 2-2242
addevent 2-97
addframe
   AVI files 2-99
addition (arithmetic operator) 2-36
addOptional
   inputParser object 2-101
addParamValue
   inputParser object 2-104
addpath 2-107
addpref function 2-109
addproperty 2-110
addRequired
   inputParser object 2-112
adding selected array elements 2-57
addsample 2-114
addsampletocollection 2-116
addtodate 2-118
addts 2-119
adjacency graph 2-908
airy 2-121
Airy functions
   relationship to modified Bessel functions 2-121
align function 2-123
aligning scattered data
   multi-dimensional 2-2195
   two-dimensional 2-1427
ALim, Axes property 2-265
all 2-127
Index

allchild function 2-129
allocation of storage (automatic) 2-3648
AlphaData
  image property 2-1591
  surface property 2-3097
  surfaceplot property 2-3118
AlphaDataMapping
  image property 2-1592
  patch property 2-2336
  surface property 2-3097
  surfaceplot property 2-3118
AmbientLightColor, Axes property 2-266
AmbientStrength
  Patch property 2-2337
  Surface property 2-3098
  surfaceplot property 2-3119
amd 2-135 2-1849
analytical partial derivatives (BVP) 2-421
analyzed function 2-2129
and 2-140
and (M-file function equivalent for &) 2-49
AND, logical
  bit-wise 2-382
angle 2-142
annotating graphs
  deleting annotations 2-145
  in plot edit mode 2-2430
annotation function 2-143
ans 2-186
anti-diagonal 2-1454
any 2-187
arccosecant 2-76
arccosine 2-66
arccotangent 2-71
arcsecant 2-218
arcsine 2-223
arctangent 2-232
  four-quadrant 2-234
arguments, M-file
  checking number of inputs 2-2186
  checking number of outputs 2-2190
  number of input 2-2188
  number of output 2-2188
  passing variable numbers of 2-3520
arithmetic operations, matrix and array
  distinguished 2-36
arithmetic operators
  reference 2-36
array
  addressing selected elements of 2-57
  displaying 2-891
  left division (arithmetic operator) 2-38
  maximum elements of 2-2061
  mean elements of 2-2066
  median elements of 2-2069
  minimum elements of 2-2101
  multiplication (arithmetic operator) 2-37
  of all ones 2-2273
  of all zeros 2-3648
  of random numbers 2-2583 2-2588
  power (arithmetic operator) 2-38
  product of elements 2-2496
  removing first n singleton dimensions
    of 2-2826
  removing singleton dimensions of 2-2917
  reshaping 2-2680
  right division (arithmetic operator) 2-37
  shift circularly 2-528
  shifting dimensions of 2-2826
  size of 2-2840
  sorting elements of 2-2854
  structure 2-1380 2-2700 2-2813
  sum of elements 2-3078
  swapping dimensions of 2-1732 2-2405
  transpose (arithmetic operator) 2-38
arrayfun 2-211
arrays
  detecting empty 2-1745
  editing 2-3616
maximum size of 2-605
opening 2-2274
arrays, structure
  field names of 2-1096
arrowhead matrix 2-592
ASCII
  delimited files
    writing 2-904
ASCII data
  converting sparse matrix after loading
    from 2-2867
    reading 2-900
    reading from disk 2-1960
    saving to disk 2-2736
ascii function 2-217
asec 2-218
asecd 2-220
asech 2-221
asin 2-223
asind 2-225
asinh 2-226
aspect ratio of axes 2-728 2-2369
assert 2-228
assignin 2-230
atan 2-232
atan2 2-234
atand 2-236
atanh 2-237
.au files
  reading 2-250
  writing 2-251
audio
  saving in AVI format 2-252
  signal conversion 2-1901 2-2169
audioplayer 1-81 2-239
audiorecorder 1-81 2-244
aufinfo 2-249
auread 2-250
AutoScale
  quivergroup property 2-2560
AutoScaleFactor
  quivergroup property 2-2560
autoselection of OpenGL 2-1133
auwrite 2-251
average of array elements 2-2066
average,running 2-1175
avi 2-252
avifile 2-252
aviinfo 2-256
aviread 2-258
axes 2-259
  editing 2-2430
  setting and querying data aspect ratio 2-728
  setting and querying limits 2-3620
  setting and querying plot box aspect
    ratio 2-2369
Aaxes
  creating 2-259
  defining default properties 2-264
  fixed-width font 2-282
  property descriptions 2-265
axis 2-303
axis crossing. See zero of a function
azimuth (spherical coordinates) 2-2883
azimuth of viewpoint 2-3537

B
BackFaceLighting
  Surface property 2-3098
  surfaceplot property 2-3119
BackFaceLightingpatch property 2-2337
BackgroundColor
  annotation textbox property 2-176
  Text property 2-3199
BackgroundColor
  Uicontrol property 2-3350
BackingStore, Figure property 2-1101
badly conditioned 2-2600
balance 2-309
Index

BarLayout
  barseries property 2-324
BarWidth
  barseries property 2-324
base to decimal conversion 2-340
base two operations
  conversion from decimal to binary 2-824
  logarithm 2-1979
  next power of two 2-2203
base2dec 2-340
Baseline
  barseries property 2-324
  stem property 2-2963
BaseValue
  areaseries property 2-196
  barseries property 2-325
  stem property 2-2963
beep 2-341
BeingDeleted
  areaseries property 2-196
  barseries property 2-325
  contour property 2-632
  errorbar property 2-974
  group property 2-1102 2-1592 2-3200
  hggroup property 2-1509
  hgtransform property 2-1529
  light property 2-1891
  line property 2-1908
  lineseries property 2-1921
  quivergroup property 2-2560
  rectangle property 2-2617
  scatter property 2-2760
  stairseries property 2-2930
  stem property 2-2963
  surface property 2-3099
  surfaceplot property 2-3120
  transform property 2-2337
  Uipushtool property 2-3430
  Uitoggletool property 2-3461
  Uitoolbar property 2-3474
Bessel functions
  first kind 2-349
  modified, first kind 2-346
  modified, second kind 2-352
  second kind 2-355
Bessel functions, modified
  relationship to Airy functions 2-121
Bessel's equation
  (defined) 2-349
  modified (defined) 2-346
besseli 2-346
besselj 2-349
besselk 2-352
bessely 2-355
beta 2-359
beta function
  (defined) 2-359
  incomplete (defined) 2-361
  natural logarithm 2-363
betainc 2-361
betaln 2-363
bicg 2-364
bicgstab 2-373
BiConjugate Gradients method 2-364
BiConjugate Gradients Stabilized method 2-373
big endian formats 2-1225
bin2dec 2-379
binary
  data
    writing to file 2-1308
  files
    reading 2-1259
    mode for opened files 2-1224
binary data
  reading from disk 2-1960
  saving to disk 2-2736
binary function 2-380
binary to decimal conversion 2-379
bisection search 2-1318
bit depth
Index

querying 2-1610
bit-wise operations
  AND 2-382
  get 2-385
  OR 2-388
  set bit 2-389
  shift 2-390
  XOR 2-392
bitand 2-382
bitcmp 2-383
bitget 2-385
bitmaps
  writing 2-1634
bitmax 2-386
bitor 2-388
bitset 2-389
bitshift 2-390
bitxor 2-392
blanks 2-393
  removing trailing 2-820
blkdiag 2-394
BMP files
  writing 2-1634
bold font
  TeX characters 2-3222
boundary value problems 2-427
box 2-395
Box, Axes property 2-267
braces, curly (special characters) 2-53
brackets (special characters) 2-53
break 2-396
breakpoints
  listing 2-769
  removing 2-757
  resuming execution from 2-760
  setting in M-files 2-773
brighten 2-397
browser
  for help 2-1494
bsxfun 2-401
buck plot (scatter function) 2-2755
Buckminster Fuller 2-3171
built in 1-70 2-400
BusyAction
  arearseries property 2-196
  Axes property 2-267
  barsers property 2-325
  contour property 2-632
  errorbar property 2-974
  Figure property 2-1102
  hggroup property 2-1509
  hgtransform property 2-1529
  Image property 2-1593
  Light property 2-1891
  Line property 2-1908 2-1921
  patch property 2-2338
  quivergroup property 2-2561
  rectangle property 2-2617
  Root property 2-2704
  scatter property 2-2760
  stairseries property 2-2930
  stem property 2-2964
  Surface property 2-3099
  surfaceplot property 2-3120
  Text property 2-3201
  Uicontextmenu property 2-3335
  Uicontrol property 2-3350
  Uimenu property 2-3396
  Uipushtool property 2-3430
  Uitoggletool property 2-3462
  Uitoolbar property 2-3474
ButtonDownFcn
  area series property 2-197
  Axes property 2-268
  barsers property 2-326
  contour property 2-633
  errorbar property 2-975
  Figure property 2-1103
  hggroup property 2-1510
  hgtransform property 2-1530

Index-5
Index

Image property 2-1593
Light property 2-1892
Line property 2-1909
lineseries property 2-1922
patch property 2-2338
quivergroup property 2-2561
rectangle property 2-2618
Root property 2-2704
scatter property 2-2761
stairseries property 2-2931
stem property 2-2964
Surface property 2-3100
surfaceplot property 2-3121
Text property 2-3201
Uicontrol property 2-3351
BVP solver properties
  analytical partial derivatives 2-421
  error tolerance 2-419
  Jacobian matrix 2-421
  mesh 2-424
  singular BVPs 2-424
  solution statistics 2-425
  vectorization 2-420
bvp4c 2-403
bvpget 2-414
bvpinit 2-415
bvpset 2-418
bvpxtend 2-427

calllib 2-429
callSoapService 2-431
camdolly 2-432
camera
  dollying position 2-432
  moving camera and target postions 2-432
  placing a light at 2-436
  positioning to view objects 2-438
  rotating around camera target 1-98 2-440
  2-442
  rotating around viewing axis 2-446
  setting and querying position 2-443
  setting and querying projection type 2-445
  setting and querying target 2-447
  setting and querying up vector 2-449
  setting and querying view angle 2-451
CameraPosition, Axes property 2-269
CameraPositionMode, Axes property 2-270
CameraTarget, Axes property 2-270
CameraTargetMode, Axes property 2-270
CameraUpVector, Axes property 2-270
CameraUpVectorMode, Axes property 2-271
CameraViewAngle, Axes property 2-271
CameraViewAngleMode, Axes property 2-271
camlight 2-436
camlookat 2-438
camorbit 2-440
campan 2-442
campos 2-443
camproj 2-445
camroll 2-446
camtarget 2-447
camup 2-449
camva 2-451
camzoom 2-453
CaptureMatrix, Root property 2-2704
CaptureRect, Root property 2-2705
cart2pol 2-454
cart2sph 2-455

caching
  MATLAB directory 2-2361
  calendar 2-428
  call history 2-2503
Callback
  Uicontextmenu property 2-3336
  Uicontrol property 2-3352
  Uimenu property 2-3397
CallbackObject, Root property 2-2704

Index-6
Cartesian coordinates 2-454 to 2-455 2-2440
   2-2883
case 2-456
   in switch statement (defined) 2-3157
   lower to upper 2-3508
   upper to lower 2-1991
cast 2-458

cat 2-459

catch 2-461

caxis 2-462

Cayley-Hamilton theorem 2-2460

CData
   Image property 2-1594
   scatter property 2-2762
   Surface property 2-3101
   surfaceplot property 2-3121
   Uicontrol property 2-3353
   Uipushtool property 2-3431
   Uitoggletool property 2-3462

CDataMapping
   Image property 2-1596
   patch property 2-2341
   Surface property 2-3102
   surfaceplot property 2-3122

CDataMode
   surfaceplot property 2-3123

CDatapatch property 2-2339

CDataSource
   scatter property 2-2762
   surfaceplot property 2-3123

cdf2rdf 2-470
cdfepoch 2-472
cdfinfo 2-473
cdfread 2-477
cdfwrite 2-481
ceil 2-484
cell 2-485

cell array

conversion to from numeric array 2-2216
creating 2-485
structure of, displaying 2-498

char 1-51 1-59 1-63 2-506

characters
   conversion, in format specification string 2-1246 2-2906
   escape, in format specification string 2-1247
      2-2906

check boxes 2-3343

Checked, Uimenu property 2-3397

checkerboard pattern (example) 2-2671

checkin 2-507
   examples 2-508
   options 2-507

checkout 2-510
   examples 2-511
   options 2-510

child functions 2-2498

Children
   areaseries property 2-198
   Axes property 2-273
   barseries property 2-327
   contour property 2-633
   errorbar property 2-975
   Figure property 2-1104
   hggroup property 2-1510
   hgtransform property 2-1530
   Image property 2-1596
   Light property 2-1892
   Line property 2-1910
   lineseries property 2-1922
   patch property 2-2341
   quivergroup property 2-2562
rectangle property 2-2619
Root property 2-2705
scatter property 2-2762
stairseries property 2-2932
stem property 2-2965
Surface property 2-3102
surfaceplot property 2-3124
Text property 2-3203
Uicontextmenu property 2-3336
Uicontrol property 2-3353
Uimenu property 2-3398
Uitoolbar property 2-3475
chol 2-513
Cholesky factorization 2-513
(as algorithm for solving linear equations) 2-2125
lower triangular factor 2-2327
minimum degree ordering and (sparse) 2-3170
preordering for 2-592
cholinc 2-517
cholupdate 2-525
circle
rectangle function 2-2612
circshift 2-528
c1a 2-529
c1abel 2-530
c1ass 2-536
class, object. See object classes
classes
field names 2-1096
loaded 2-1659
c1c 2-538 2-545
c1ear 2-539
serial port I/O 2-544
c1earing
Command Window 2-538
items from workspace 2-539
Java import list 2-541
c1f 2-545
ClickedCallback
Uipushtool property 2-3431
Uitoggletool property 2-3463
CLim, Axes property 2-273
CLimMode, Axes property 2-274
clipboard 2-546
Clipping
areaeriseries property 2-198
Axes property 2-274
barseries property 2-327
contour property 2-634
errobar property 2-976
Figure property 2-1104
hgggroup property 2-1511
hgttransform property 2-1531
Image property 2-1597
Light property 2-1892
Line property 2-1910
lineseries property 2-1923
quivergroup property 2-2562
rectangle property 2-2619
Root property 2-2705
scatter property 2-2763
stairseries property 2-2932
stem property 2-2965
Surface property 2-3102
surfaceplot property 2-3124
Text property 2-3203
Uicontrol property 2-3353
Clippingpatch property 2-2341
clock 2-547
close 2-548
AVI files 2-550
close (ftp) function 2-551
CloseRequestFcn, Figure property 2-1104
closest point search 2-924
closest triangle search 2-3298
closingiles 2-1059
MATLAB 2-2551
cmapeditor 2-572
cmopts 2-553
code
  analyzer 2-2129
colamd 2-555
colmmd 2-559
colon operator 2-57
Color
  annotation arrow property 2-147
  annotation doublearrow property 2-151
  annotation line property 2-159
  annotation textbox property 2-176
  Axes property 2-274
  errorbar property 2-976
  Figure property 2-1107
  Light property 2-1892
  Line property 2-1911
  lineseries property 2-1923
  quivergroup property 2-2562
  stairseries property 2-2932
  stem property 2-2966
  Text property 2-3203
  textarrow property 2-165
color of fonts, see also FontColor property 2-3222
colorbar 2-561
colormap 2-567
  editor 2-572
Colormap, Figure property 2-1107
colormaps
  converting from RGB to HSV 1-97 2-2690
  plotting RGB components 1-97 2-2691
ColorOrder, Axes property 2-274
ColorSpec 2-590
colperm 2-592
COM

object methods
  actxcontrol 2-81
  actxcontrollist 2-88
  actxcontrolselect 2-89
  actxserver 2-93
  addproperty 2-110
  delete 2-850
  deleteproperty 2-856
  eventlisteners 2-1002
  events 2-1004
  get 1-111 2-1363
  inspect 2-1675
  invoke 2-1729
  iscom 2-1743
  isevent 2-1753
  isinterface 2-1765
  ismethod 2-1774
  isprop 2-1795
  load 2-1965
  move 2-2150
  propedit 2-2506
  registerevent 2-2660
  release 2-2665
  save 2-2744
  send 2-2789
  set 1-112 2-2799
  unregisterallevents 2-3492
  unregisterevent 2-3495
server methods
  Execute 2-1006
  Feval 2-1068
combinations of n elements 2-2194
combs 2-2194
comet 2-594
comet3 2-596
comma (special characters) 2-55
command syntax 2-1490 2-3176
Command Window
  clearing 2-538
  cursor position 1-4 2-1550
Index

get width 2-599
commandhistory 2-598
commands
  help for 2-1489 2-1499
  system 1-4 1-11 2-3179
  UNIX 2-3488
commandwindow 2-599
comments
  block of 2-55
common elements. See set operations, intersection
compan 2-600
companion matrix 2-600
compass 2-601
complementary error function
  (defined) 2-965
  scaled (defined) 2-965
complete elliptic integral
  (defined) 2-949
  modulus of 2-947 2-949
complex 2-603 2-1583
  exponential (defined) 2-1014
  logarithm 2-1976 to 2-1977
  numbers 2-1559
  numbers, sorting 2-2854 2-2858
  phase angle 2-142
  sine 2-2834
  unitary matrix 2-2530
See also imaginary
complex conjugate 2-617
  sorting pairs of 2-691
complex data
  creating 2-603
complex numbers, magnitude 2-59
complex Schur form 2-2776
compression
  lossy 2-1638
computer 2-605
computer MATLAB is running on 2-605
concatenation

of arrays 2-459
cond 2-607
condeig 2-608
condest 2-609
condition number of matrix 2-607 2-2600
  improving 2-309
coneplot 2-611
conj 2-617
conjugate, complex 2-617
  sorting pairs of 2-691
connecting to FTP server 2-1288
contents.m file 2-1490
context menu 2-3332
continuation (..., special characters) 2-55
continue 2-618
continued fraction expansion 2-2594
contour
  and mesh plot 2-1034
  filled plot 2-1026
  functions 2-1022
  of mathematical expression 2-1023
  with surface plot 2-1052
contour3 2-624
contourc 2-627
contourf 2-629
ContourMatrix
  contour property 2-634
contours
  in slice planes 2-651
contourslice 2-651
contrast 2-655
conv 2-656
conv2 2-658
conversion
  base to decimal 2-340
  binary to decimal 2-379
  Cartesian to cylindrical 2-454
  Cartesian to polar 2-454
  complex diagonal to real block diagonal 2-470
  cylindrical to Cartesian 2-2440
decimal number to base 2-817 2-823
decimal to binary 2-824
decimal to hexadecimal 2-825
full to sparse 2-2864
hexadecimal to decimal 2-1503
integer to string 2-1689
lowercase to uppercase 2-3508
matrix to string 2-2031
numeric array to cell array 2-2216
numeric array to logical array 2-1980
numeric array to string 2-2218
partial fraction expansion to
pole-residue 2-2682
polar to Cartesian 2-2440
pole-residue to partial fraction
expansion 2-2682
real to complex Schur form 2-2733
spherical to Cartesian 2-2883
string matrix to cell array 2-500
string to numeric array 2-2987
uppercase to lowercase 2-1991
vector to character string 2-506
conversion characters in format specification
string 2-1246 2-2906
convex hulls
    multidimensional visualization 2-667
two-dimensional visualization 2-664
convhull 2-664
convhulln 2-667
convn 2-670
convolution 2-656
    inverse. See deconvolution
two-dimensional 2-658
coordinate system and viewpoint 2-3537
coordinates
    Cartesian 2-454 to 2-455 2-2440 2-2883
cylindrical 2-454 to 2-455 2-2440
polar 2-454 to 2-455 2-2440
spherical 2-2883
coordinates. 2-454
See also conversion
copyfile 2-671
copyobj 2-674
corrcorrcoef 2-676
cos 2-679
cosd 2-681
cosecant
    hyperbolic 2-702
    inverse 2-76
    inverse hyperbolic 2-79
cosh 2-682
cosine 2-679
    hyperbolic 2-682
    inverse 2-66
    inverse hyperbolic 2-69
cot 2-684
cotangent 2-684
    hyperbolic 2-687
    inverse 2-71
    inverse hyperbolic 2-74
cotd 2-686
coth 2-687
cov 2-689
cplxpair 2-691
cputime 2-692
createClassFromWsdl 2-693
creaticopy
    inputParser object 2-695
CreateFcn
    areasereseries property 2-198
Axes property 2-275
barserseries property 2-327
contour property 2-635
errorbar property 2-976
Figure property 2-1108
group property 2-1531
hggroup property 2-1511
Image property 2-1597
Light property 2-1893
Line property 2-1911
Index

lineseries property 2-1923
patch property 2-2341
quivergroup property 2-2563
rectangle property 2-2619
Root property 2-2705
scatter property 2-2763
stairseries property 2-2932
stemseries property 2-2966
Surface property 2-3103
surfaceplot property 2-3124
Text property 2-3203
Uicontextmenu property 2-3336
Uicontrol property 2-3353
Uimenu property 2-3398
Uipushtool property 2-3432
Uitoggletool property 2-3463
Uitoolbar property 2-3475
createSoapMessage 2-697
creating your own MATLAB functions 2-1294
cross 2-698
cross product 2-698
csc 2-699
cscd 2-701
csch 2-702
csvread 2-704
csvwrite 2-707
ctranspose (M-file function equivalent for \(q\) 2-42
ctranspose (timeseries) 2-709
cubic interpolation 2-1705 2-1708 2-1711 2-2379
  piecewise Hermite 2-1695
cubic spline interpolation
  one-dimensional 2-1695 2-1705 2-1708 2-1711
cumprod 2-711
cumsum 2-713
cumtrapz 2-714
cumulative
  product 2-711
  sum 2-713
curl 2-716
curly braces (special characters) 2-53
current directory 2-2523
  changing 2-467
CurrentAxes 2-1109
CurrentAxes, Figure property 2-1109
CurrentCharacter, Figure property 2-1109
CurrentFigure, Root property 2-2705
CurrentMenu, Figure property (obsolete) 2-1109
CurrentObject, Figure property 2-1110
CurrentPoint
  Axes property 2-276
  Figure property 2-1110
cursor images
  reading 2-1622
cursor position 1-4 2-1550
Curvature, rectangle property 2-2620
curve fitting (polynomial) 2-2452
customverctrl 2-719
Cuthill-McKee ordering, reverse 2-3160 2-3171
cylinder 2-720
cylindrical coordinates 2-454 to 2-455 2-2440

D
daqread 2-723
daspect 2-728
data
  ASCII
    reading from disk 2-1960
  ASCII, saving to disk 2-2736
  binary
    writing to file 2-1308
  binary, saving to disk 2-2736
  computing 2-D stream lines 1-101 2-2994
  computing 3-D stream lines 1-101 2-2996
  formatted
    reading from files 2-1275
    writing to file 2-1245
    formatting 2-1245 2-2904
isosurface from volume data 2-1788
reading binary from disk 2-1960
reading from files 2-3228
reducing number of elements in 1-101 2-2635
smoothing 3-D 1-101 2-2852
writing to strings 2-2904
data aspect ratio of axes 2-728
data types
  complex 2-603
data, aligning scattered
    multi-dimensional 2-2195
two-dimensional 2-1427
data, ASCII
  converting sparse matrix after loading from 2-2867
DataAspectRatio, Axes property 2-278
DataAspectRatioMode, Axes property 2-281
datatipinfo 2-736
date 2-737
date and time functions 2-960
date string
  format of 2-742
date vector 2-754
datenum 2-738
datestr 2-742
datevec 2-753
dbclear 2-757
dbcont 2-760
dbdow 2-761
dblquad 2-762
dbmex 2-764
dbquerit 2-765
dbstack 2-767
dbstatus 2-769
dbstop 2-773
dbtype 2-783
dbup 2-784
DDE solver properties
  error tolerance 2-805
event location 2-811
solver output 2-807
step size 2-809
dde23 2-785
ddeadv 1-112 2-790
ddeexec 2-792
ddeget 2-793
ddeinit 1-112 2-794
ddephas2 output function 2-808
ddephas3 output function 2-808
ddep1ot output function 2-808
ddepoke 2-795
ddeprint output function 2-808
ddeque 2-797
ddesd 2-799
ddeset 2-804
ddterm 2-815
ddeunadv 2-816
deal 2-817
debblank 2-820
debugging
  changing workspace context 2-761
  changing workspace to calling M-file 2-784
  displaying function call stack 2-767
  M-files 2-1836 2-2498
  MEX-files on UNIX 2-764
  removing breakpoints 2-757
  resuming execution from breakpoint 2-771
  setting breakpoints in 2-773
  stepping through lines 2-771
dec2base 2-817 2-823
dec2bin 2-824
dec2hex 2-825
decic function 2-826
decimal number to base conversion 2-817 2-823
decimal point (.)
  (special characters) 2-54
  to distinguish matrix and array operations 2-36
decomposition
Dulmage-Mendelsohn 2-908
"economy-size" 2-2530 2-3149
orthogonal-triangular (QR) 2-2530
Schur 2-2776
singular value 2-2593 2-3149
deconv 2-828
deconvolution 2-828
definite integral 2-2542
del operator 2-829
de12 2-829
delaunay 2-832
Delaunay tessellation
  3-dimensional visualization 2-839
  multidimensional visualization 2-843
Delaunay triangulation
  visualization 2-832
delaunay3 2-839
delaunayn 2-843
delete 2-848 2-850
  serial port I/O 2-853
  timer object 2-855
delete (ftp) function 2-852
DeleteFcn
  areaseres property 2-199
  Axes property 2-281
  barseries property 2-328
  contour property 2-635
  errorbar property 2-976
  Figure property 2-1112
  hggroup property 2-1512
  hgtransform property 2-1532
  Image property 2-1597
  Light property 2-1894
  lineseries property 2-1924
  quivergroup property 2-2563
  Root property 2-2706
  scatter property 2-2764
  stairseries property 2-2933
  stem property 2-2967
  Surface property 2-3103
surfaceplot property 2-3125
Text property 2-3204 2-3206
Uicontextmenu property 2-3337 2-3354
Uimenu property 2-3399
Upushtool property 2-3433
Uitoggletool property 2-3464
Uitoolbar property 2-3476
DeleteFcn, line property 2-1912
DeleteFcn, rectangle property 2-2621
DeleteFcnpatch property 2-2342
deleteproperty 2-856
deleting
  files 2-848
  items from workspace 2-539
delevent 2-858
delimiters in ASCII files 2-900 2-904
delsample 2-859
delsamplefromcollection 2-860
delsamplefromcollection 2-860
demo 2-861
demos
  in Command Window 2-927
density
  of sparse matrix 2-2204
depdir 2-866
dependence, linear 2-3070
dependent functions 2-2498
depfun 2-867
derivative
  approximate 2-882
  polynomial 2-2449
det 2-871
detecting
  alphabetic characters 2-1769
  empty arrays 2-1745
  global variables 2-1759
  logical arrays 2-1770
  members of a set 2-1772
  objects of a given class 2-1737
  positive, negative, and zero array
    elements 2-2833
sparse matrix 2-1804
determinant of a matrix 2-871
detrend 2-872
detrend (timeseries) 2-874
deval 2-875
diag 2-877
diagonal 2-877
  anti- 2-1454
  k-th (illustration) 2-3283
  main 2-877
  sparse 2-2869
dialog 2-879
dialog box
  error 2-990
  help 2-1497
  input 2-1664
  list 2-1955
  message 2-2163
  print 1-91 1-103 2-2487
  question 1-103 2-2549
  warning 2-3561
diary 2-880
Diary, Root property 2-2706
DiaryFile, Root property 2-2706
diff 2-882
differences
  between adjacent array elements 2-882
  between sets 2-2811
differential equation solvers
  defining an ODE problem 2-2245
  ODE boundary value problems 2-403
    adjusting parameters 2-418
    extracting properties 2-414
    extracting properties of 2-994 to 2-995
    2-3280 to 2-3281
    forming initial guess 2-415
  ODE initial value problems 2-2231
    adjusting parameters of 2-2252
    extracting properties of 2-2251
  parabolic-elliptic PDE problems 2-2387
diffuse 2-884
DiffuseStrength
  Surface property 2-3104
    surfaceplot property 2-3125
  DiffuseStrengthpatch property 2-2343
digamma function 2-2508
dimension statement (lack of in MATLAB) 2-3648
dimensions
  size of 2-2840
Diophantine equations 2-1348
dir 2-885
dir (ftp) function 2-888
direct term of a partial fraction expansion 2-2682
directories 2-467
  adding to search path 2-107
  checking existence of 2-1009
  copying 2-671
  creating 2-2112
  listing contents of 2-885
  listing MATLAB files in 2-3587
  listing, on UNIX 2-1992
MATLAB
  caching 2-2361
  removing 2-2696
  removing from search path 2-2701
See also directory, search path
directory 2-885
  changing on FTP server 2-469
  listing for FTP server 2-888
  making on FTP server 2-2115
  MATLAB location 2-2042
  root 2-2042
  temporary system 2-3187
See also directories
directory, changing 2-467
directory, current 2-2523
disconnect 2-551
discontinuities, eliminating (in arrays of phase angles) 2-3504
discontinuities, plotting functions with 2-1050
 discontinuous problems 2-1222
disp 2-891
 memmapfile object 2-2072
 serial port I/O 2-893
 timer object 2-894
display 2-896
display format 2-1232
displaying output in Command Window 2-2148
DisplayName
 areaseries property 2-199
 barseries property 2-328
 contour property 2-636
 errorbar property 2-977
 lineseries property 2-1924
 quivergroup property 2-2564
 scatter property 2-2764
 stairseries property 2-2934
 stem property 2-2967
distribution
 Gaussian 2-965
 Dithermap 2-1113
 DithermapMode, Figure property 2-1113
division
 array, left (arithmetic operator) 2-38
 array, right (arithmetic operator) 2-37
 by zero 2-1652
 matrix, left (arithmetic operator) 2-37
 matrix, right (arithmetic operator) 2-37
 of polynomials 2-828
divisor
 greatest common 2-1348
dll libraries
MATLAB functions
 calllib 2-429
 libfunctions 2-1874
 libfunctionsview 2-1876
 libisloaded 2-1878
 libpointer 2-1880
 libstruct 2-1882
 loadlibrary 2-1968
 unloadlibrary 2-3490
dlmread 2-900
dlmwrite 2-904
dmperm 2-908
Dockable, Figure property 2-1113
docsearch 2-913
documentation
 displaying online 2-1494
dolly camera 2-432
dos 2-915
 UNC pathname error 2-916
dot 2-917
dot product 2-698 2-917
dot-parentheses (special characters 2-55
double 1-58 2-918
double click, detecting 2-1136
double integral
 numerical evaluation 2-762
DoubleBuffer, Figure property 2-1113
downloading files from FTP server 2-2100
dragrect 2-919
drawing shapes
 circles and rectangles 2-2612
DrawMode, Axes property 2-281
drawnow 2-921
dsearch 2-923
dsearchn 2-924
Dulmage-Mendelsohn decomposition 2-908
dynamic fields 2-55
E

echo 2-925
Echo, Root property 2-2706
echodem 2-927
dge finding, Sobel technique 2-660
EdgeAlpha
  patch property 2-2343
  surface property 2-3104
  surfaceplot property 2-3126
EdgeColor
  annotation ellipse property 2-156
  annotation rectangle property 2-162
  annotation textbox property 2-176
  areaperseries property 2-200
  barseries property 2-329
  patch property 2-2343
  Surface property 2-3105
  surfaceplot property 2-3126
  Text property 2-3205
EdgeColor, rectangle property 2-2622
End
  patch property 2-2344
  Surface property 2-3106
  surfaceplot property 2-3127
end caps for isosurfaces 2-1778
end of line, indicating 2-55
end-of-file indicator 2-1064
eomday 2-960
eps 2-961
eq 2-963
equal arrays
  detecting 2-1748 2-1751
equal sign (special characters) 2-54
equations, linear
  accuracy of solution 2-607
EraseMode
  areaperseries property 2-200
  Wilkinson test matrix and 2-3607
eigenvalues
  effect of roundoff error 2-309
  improving accuracy 2-309
eigenvector
  left 2-932
  matrix, generalized 2-2580
  right 2-932
eigs 2-937
elevation (spherical coordinates) 2-2883
elevation of viewpoint 2-3537
ellipj 2-947
ellipke 2-949
ellipsoid 1-89 2-951
elliptic functions, Jacobian
  (defined) 2-947
elliptic integral
  complete (defined) 2-949
  modulus of 2-947 2-949
else 2-953
elseif 2-954
Enable
  Uicontrol property 2-3355
  Uimenu property 2-3400
  Uipushtool property 2-3433
  Uitogglehtool property 2-3465
else 2-953
elseif 2-954
Enable
barseries property 2-329
contour property 2-636
errorbar property 2-977
hggroupproperty 2-1512
hgttransform property 2-1532
Image property 2-1598
Line property 2-1913
lineseries property 2-1924
quivergroup property 2-2564
rectangle property 2-2622
scatter property 2-2764
stairseries property 2-2934
stem property 2-2967
Surface property 2-3106
surfaceplot property 2-3127
Text property 2-3207
EraseMode property 2-2345
error 2-967
roundoff. See roundoff error
error function
  complementary 2-965
  (defined) 2-965
  scaled complementary 2-965
error message
  displaying 2-967
  Index into matrix is negative or zero 2-1981
  retrieving last generated 2-1839 2-1846
error messages
  Out of memory 2-2308
error tolerance
  BVP problems 2-419
  DDE problems 2-805
  ODE problems 2-2253
errorbars 2-971
erroldlg 2-990
ErrorMessage, Root property 2-2706
errors
  in file input/output 2-1065
ErrorType, Root property 2-2707
escape characters in format specification
  string 2-1247 2-2906
etime 2-993
etree 2-994
etreeplot 2-995
eval 2-996
evalc 2-999
evalin 2-1000
event location (DDE) 2-811
event location (ODE) 2-2260
eventlisteners 2-1002
events 2-1004
examples
  calculating isosurface normals 2-1785
  contouring mathematical expressions 2-1023
  isosurface end caps 2-1778
  isosurfaces 2-1789
  mesh plot of mathematical function 2-1032
  mesh/contour plot 2-1036
  plotting filled contours 2-1027
  plotting function of two variables 2-1040
  plotting parametric curves 2-1043
  polar plot of function 2-1046
  reducing number of patch faces 2-2632
  reducing volume data 2-2635
  subsampling volume data 2-3075
  surface plot of mathematical function 2-1050
  surface/contour plot 2-1054
Excel spreadsheets
  loading 2-3625
exclamation point (special characters) 2-56
Execute 2-1006
executing statements repeatedly 2-1230 2-3594
execution
  improving speed of by setting aside
    storage 2-3648
  pausing M-file 2-2367
  resuming from breakpoint 2-760
  time for M-files 2-2498
exifread 2-1008
exist 2-1009
exit 2-1013
exp 2-1014
expint 2-1015
expm 2-1016
expm1 2-1018
exponential 2-1014
    complex (defined) 2-1014
    integral 2-1015
    matrix 2-1016
exponentiation
    array (arithmetic operator) 2-38
    matrix (arithmetic operator) 2-38
export2wsdlg 2-1019
extension, filename
    .m 2-1294
    .mat 2-2736
Extent
    Text property 2-3208
    Uicontrol property 2-3356
eye 2-1021
ezcontour 2-1022
ezcontourf 2-1026
ezmesh 2-1030
ezmeshc 2-1034
ezplot 2-1038
ezplot3 2-1042
ezpolar 2-1045
ezsurf 2-1048
ezsurfcl 2-1052
annotation ellipse property 2-156
annotation rectangle property 2-162
areaseries property 2-201
barseries property 2-330
Surface property 2-3108
surfaceplot property 2-3129
FaceColor, rectangle property 2-2623
FaceColorpatch property 2-2346
FaceLighting
    Surface property 2-3109
    surfaceplot property 2-3130
FaceLightingpatch property 2-2347
faces, reducing number in patches 1-101 2-2631
Faces,patch property 2-2347
FaceVertexAlphaData, patch property 2-2348
FaceVertexCData,patch property 2-2349
factor 2-1056
factorial 2-1057
factorization 2-2530
    LU 2-2008
    QZ 2-2451 2-2580
        See also decomposition
factorization, Cholesky 2-513
        (as algorithm for solving linear
        equations) 2-2125
        minimum degree ordering and
        (sparse) 2-3170
        preordering for 2-592
factors, prime 2-1056
false 2-1058
fclose 2-1059
    serial port I/O 2-1060
feather 2-1062
feof 2-1064
terror 2-1065
feval 2-1066
Feval 2-1068
fft 2-1073
FFT. See Fourier transform
fft2 2-1078

F

F-norm 2-2207
FaceAlpha
    annotation textbox property 2-177
FaceAlphapatch property 2-2346
FaceAlphasurface property 2-3108
FaceAlphasurfaceplot property 2-3129
FaceColor
Index

fftn 2-1079
fftshift 2-1081
fftw 2-1083
FFTW 2-1076
fgetl 2-1088
serial port I/O 2-1089
fgets 2-1092
serial port I/O 2-1093
field names of a structure, obtaining 2-1096
fieldname 2-1096
fields, noncontiguous, inserting data into 2-1308
fields, of structures
  dynamic 2-55
fig files
  annotating for printing 2-1256
figure 2-1098
Figure
  creating 2-1098
    defining default properties 2-1100
    properties 2-1101
    redrawing 1-95 2-2638
figure windows, displaying 2-1188
figurepalette 1-86 2-1153
figures
  annotating 2-2430
  opening 2-2274
  saving 2-2747
Figures
  updating from M-file 2-921
file
  extension, getting 2-1165
  modification date 2-885
  position indicator
    finding 2-1287
    setting 2-1285
    setting to start of file 2-1274
file formats
  getting list of supported formats 2-1612
  reading 2-723 2-1620
  writing 2-1633
file size
  querying 2-1610
fileattrib 2-1155
filebrowser 2-1161
filehandle 2-1166
filemarker 2-1164
filename
  building from parts 2-1291
    parts 2-1165
    temporary 2-3188
filename extension
  .m 2-1294
  .mat 2-2736
fileparts 2-1165
files 2-1059
  ASCII delimited
    reading 2-900
    writing 2-904
beginning of, rewinding to 2-1274 2-1617
checking existence of 2-1009
closing 2-1059
contents, listing 2-3306
copying 2-671
deleting 2-848
deleting on FTP server 2-852
end of, testing for 2-1064
errors in input or output 2-1065
Excel spreadsheets
  loading 2-3625
fig 2-2747
figure, saving 2-2747
finding position within 2-1287
going next line 2-1088
going next line (with line terminator) 2-1092
listing
  in directory 2-3587
    names in a directory 2-885
listing contents of 2-3306
locating 2-3591
mdl  2-2747
mode when opened  2-1224
model, saving  2-2747
opening  2-1225  2-2274
    in Web browser  1-5  1-8  2-3581
opening in Windows applications  2-3608
path, getting  2-1165
pathname for  2-3591
reading
    binary  2-1259
    data from  2-3228
    formatted  2-1275
reading data from  2-723
reading image data from  2-1620
rewinding to beginning of  2-1274  2-1617
setting position within  2-1285
size, determining  2-887
sound
    reading  2-250  2-3575
    writing  2-251 to  2-252  2-3580
startup  2-2040
version, getting  2-1165
.wav
    reading  2-3575
    writing  2-3580
WK1
    loading  2-3612
    writing to  2-3614
writing binary data to  2-1308
writing formatted data to  2-1245
writing image data to  2-1633
See also file
    filesep  2-1167
    fill  2-1168
    Fill
        contour property  2-637
    fill3  2-1171
    filter  2-1174
        digital  2-1174
        finite impulse response (FIR)  2-1174
infinite impulse response (IIR)  2-1174
two-dimensional  2-658
filter (timeseries)  2-1177
filter2  2-1180
find  2-1182
findall function  2-1187
findfigs  2-1188
finding  2-1182
    sign of array elements  2-2833
    zero of a function  2-1314
See also detecting
findobj  2-1189
findstr  2-1192
finish  2-1193
finish.m  2-2551
FIR filter  2-1174
FitheightToText
    annotation textbox property  2-177
fitsinfo  2-1194
fitsread  2-1203
fix  2-1205
fixed-width font
    axes  2-282
    text  2-3209
    uicontrols  2-3357
FixedColors, Figure property  2-1114
FixedWidthFontName, Root property  2-2707
flints  2-2169
flipdim  2-1206
fliplr  2-1207
flipud  2-1208
floating-point
    integer, maximum  2-386
floating-point arithmetic, IEEE
    smallest positive number  2-2607
floor  2-1210
flops  2-1211
flow control
    break  2-396
    case  2-456
end 2-958
error 2-968
for 2-1230
keyboard 2-1836
otherwise 2-2307
return 2-2689
switch 2-3157
while 2-3594
fminbnd 2-1213
fminsearch 2-1218
font
fixed-width, axes 2-282
fixed-width, text 2-3209
fixed-width, uicontrols 2-3357
FontAngle
annotation textbox property 2-179
Axes property 2-282
Text property 2-3209
Uicontrol property 2-3356
FontName
annotation textbox property 2-179
Axes property 2-282
Text property 2-3209
textarrow property 2-166
Uicontrol property 2-3357
fonts
bold 2-166 2-179 2-3210
italic 2-166 2-179 2-3209
specifying size 2-3209
TeX characters
bold 2-3222
italics 2-3222
specifying family 2-3222
specifying size 2-3222
units 2-166 2-179 2-3210
FontSize
annotation textbox property 2-179
Axes property 2-283
Text property 2-3209
textarrow property 2-166
Uicontrol property 2-3357
Uicontrol property 2-3357
FontUnits
Axes property 2-283
Text property 2-3210
Uicontrol property 2-3358
FontWeight
annotation textbox property 2-179
Axes property 2-284
Text property 2-3210
textarrow property 2-166
Uicontrol property 2-3358
fopen 2-1223
serial port I/O 2-1228
for 2-1230
ForeColor
Uicontrol property 2-3358
Uimenu property 2-3400
format 2-1232
precision when writing 2-1259
reading files 2-1276
specification string, matching file data
to 2-2921
Format 2-2707
formats
big endian 2-1225
little endian 2-1225
FormatSpacing, Root property 2-2708
formatted data
reading from file 2-1275
writing to file 2-1245
formatting data 2-2904
Fourier transform
algorithm, optimal performance of 2-1076
2-1569 2-1571 2-2203
as method of interpolation 2-1710
convolution theorem and 2-656
discrete, n-dimensional 2-1079
discrete, one-dimensional 2-1073
discrete, two-dimensional 2-1078
fast 2-1073
inverse, n-dimensional 2-1573
inverse, one-dimensional 2-1569
inverse, two-dimensional 2-1571
shifting the zero-frequency component of 2-1082
fplot 2-1240 2-1255
fprintf 2-1245
    displaying hyperlinks with 2-1250
    serial port I/O 2-1252
fraction, continued 2-2594
fragmented memory 2-2308
frame2im 2-1255
frames 2-3343
frames for printing 2-1256
fread 2-1259
    serial port I/O 2-1269
freqspace 2-1273
frequency response
    desired response matrix
    frequency spacing 2-1273
frequency vector 2-1988
frewind 2-1274
fscanf 2-1275
    serial port I/O 2-1281
fseek 2-1285
ftell 2-1287
FTP
    connecting to server 2-1288
ftp function 2-1288
full 2-1290
fullfile 2-1291
func2str 2-1292
function 2-1294
function handle 2-1296
function handles
    overview of 2-1296
function syntax 2-1490 2-3176
functions 2-1299
    call history 2-2503
    call stack for 2-767
checking existence of 2-1009
clearing from workspace 2-539
finding using keywords 2-1989
help for 2-1489 2-1499
in memory 2-1659
locating 2-3591
pathname for 2-3591
that work down the first non-singleton dimension 2-2826
funm 2-1303
fwrite 2-1308
    serial port I/O 2-1310
fzero 2-1314

g

gallery 2-1320
gamma function
    (defined) 2-1343
    incomplete 2-1343
    logarithm of 2-1343
    logarithmic derivative 2-2508
Gaussian distribution function 2-965
Gaussian elimination
    (as algorithm for solving linear equations) 2-1725 2-2126
    Gauss Jordan elimination with partial pivoting 2-2731
    LU factorization 2-2008
gca 2-1345
gcbf function 2-1346
gcbo function 2-1347
gcd 2-1348
gcf 2-1350
gco 2-1351
ge 2-1352
generalized eigenvalue problem 2-932 2-2450
 generating a sequence of matrix names (M1 through M12) 2-997
genpath 2-1354
genvarname 2-1356
geodesic dome 2-3171
get 1-111 2-1360 2-1363
memmapfile object 2-2073
serial port I/O 2-1365
timer object 2-1367
get (timeseries) 2-1369
get (tscollection) 2-1370
getabstime (timeseries) 2-1371
getabstime (tscollection) 2-1373
getappdata function 2-1375
getdatasamplesize 2-1378
getenv 2-1379
getfield 2-1380
getframe 2-1382
image resolution and 2-1383
getinterpmethod 2-1388
getpixelposition 2-1389
getpref function 2-1391
getqualitydesc 2-1393
getsamplesusingtime (timeseries) 2-1394
getsamplesusingtime (tscollection) 2-1395
gettimeseriesnames 2-1396
getsafteratevent 2-1397
getsafeterevent 2-1398
getsatevent 2-1399
getsbeforeatevent 2-1400
getsbeforeerevent 2-1401
getsbetweenevents 2-1402
GIF files
writing 2-1634
ginput function 2-1407
global 2-1409
global variable
defining 2-1409
global variables, clearing from workspace 2-539
gmres 2-1411
golden section search 2-1216
Goup
defining default properties 2-1527
gplot 2-1417
grabcode function 2-1419
gradient 2-1421
gradient, numerical 2-1421
graph
adjacency 2-908
graphics objects
Axes 2-259
Figure 2-1098
getting properties 2-1360
Image 2-1584
Light 2-1889
Line 2-1902
Patch 2-2328
resetting properties 1-99 2-2679
Root 1-93 2-2703
setting properties 1-93 1-95 2-2795
Surface 1-93 1-96 2-3092
Text 1-93 2-3194
uicontextmenu 2-3332
Uicontrol 2-3342
Uimenu 1-106 2-3392
graphics objects, deleting 2-848
graphs
editing 2-2430
graymon 2-1424
greatest common divisor 2-1348
Greek letters and mathematical symbols 2-170
2-182 2-3220
grid 2-1425
aligning data to a 2-1427
grid arrays
for volumetric plots 2-2090
multi-dimensional 2-2195
griddata 2-1427
griddata3 2-1431
griddatan 2-1434
GridLineStyle, Axes property 2-284
group
hgroup function 2-1506
gsvd 2-1437
gt 2-1443
gtext 2-1445
guidata function 2-1446
guihandles function 2-1449
GUIs, printing 2-2482
gunzip 2-1450 2-1452

H
H1 line 2-1491 to 2-1492
hadamard 2-1453
Hadamard matrix 2-1453
subspaces of 2-3070
handle graphics
hgtransform 2-1523
handle graphics group 2-1506
HandleVisibility
areaseries property 2-202
Axes property 2-284
barseries property 2-331
contour property 2-638
errorbar property 2-978
Figure property 2-1114
hggroup property 2-1514
hgtransform property 2-1534
Image property 2-1599
Light property 2-1894
Line property 2-1914
lineseries property 2-1926
patch property 2-2351
quivergroup property 2-2565
rectangle property 2-2623
Root property 2-2708
stairseries property 2-2935
stem property 2-2969
Surface property 2-3109
surfaceplot property 2-3131
Text property 2-3210
Uicontextmenu property 2-3338
Uicontrol property 2-3358
Uimenu property 2-3400
Uipushtool property 2-3434
Uitoggletool property 2-3465
Uitoolbar property 2-3477
hankel 2-1454
Hankel matrix 2-1454
HDF
appending to when saving
(WriteMode) 2-1638
compression 2-1637
setting JPEG quality when writing 2-1638
HDF files
writing images 2-1634
HDF4
summary of capabilities 2-1455
HDF5
high-level access 2-1457
summary of capabilities 2-1457
HDF5 class
low-level access 2-1457
hdf5info 2-1460
hdf5read 2-1462
hdf5write 2-1464
hdfinfo 2-1468
hdfread 2-1476
hdftool 2-1488
Head1Length
annotation doublearrow property 2-151
Head1Style
annotation doublearrow property 2-152
Head1Width
annotation doublearrow property 2-153
Head2Length
annotation doublearrow property 2-151
Head2Style
annotation doublearrow property 2-152
Head2Width
annotation doublearrow property 2-153
HeadLength
Index

annotation arrow property 2-147
  textarrow property 2-167
HeadStyle
  annotation arrow property 2-147
  textarrow property 2-167
HeadWidth
  annotation arrow property 2-148
  textarrow property 2-168
Height
  annotation ellipse property 2-157
help 2-1489
  contents file 2-1490
  creating for M-files 2-1491
  keyword search in functions 2-1989
  online 2-1489
Help browser 2-1494
  accessing from doc 2-910
Help Window 2-1499
helpbrowser 2-1494
helpdesk 2-1496
helpdlg 2-1497
helpwin 2-1499
Hermite transformations, elementary 2-1348
hess 2-1500
Hessenberg form of a matrix 2-1500
hex2dec 2-1503
hex2num 2-1504
hidden 2-1539
Hierarchical Data Format (HDF) files
  writing images 2-1634
hilb 2-1540
Hilbert matrix 2-1540
  inverse 2-1728
hist 2-1541
histc 2-1545
HitTest
  areasereseries property 2-203
  Axes property 2-285
  barseries property 2-332
  contour property 2-639
errorbar property 2-980
  Figure property 2-1116
  hggroup property 2-1515
  hgtransform property 2-1535
  Image property 2-1601
  Light property 2-1896
  Line property 2-1914
  lineseries property 2-1927
  Patch property 2-2352
  quivergroup property 2-2567
  rectangle property 2-2625
  Root property 2-2708
  scatter property 2-2767
  stairseries property 2-2937
  stem property 2-2970
  Surface property 2-3111
  surfaceplot property 2-3132
  Text property 2-3211
  Uicontrol property 2-3359
HitTestArea
  areasereseries property 2-204
  barseries property 2-333
  contour property 2-639
  errorbar property 2-980
  quivergroup property 2-2567
  scatter property 2-2768
  stairseries property 2-2937
  stem property 2-2970
hold 2-1548
home 2-1550
HorizontalAlignment
  Text property 2-3212
  textarrow property 2-168
  textbox property 2-179
  Uicontrol property 2-3360
horzcat 2-1551
horzcat (M-file function equivalent for []) 2-56
horzcat (tscollection) 2-1553
hostid 2-1554
Householder reflections (as algorithm for solving linear equations) 2-2127
hsv2rgb 2-1555
HTML
  in Command Window 2-2035
  save M-file as  2-2511
HTML browser
  in MATLAB 2-1494
HTML files
  opening  1-5 1-8 2-3581
hyperbolic
  cosecant 2-702
  cosecant, inverse 2-79
  cosine 2-682
  cosine, inverse 2-69
  cotangent 2-687
  cotangent, inverse 2-74
  secant 2-2783
  secant, inverse 2-221
  sine 2-2838
  sine, inverse 2-226
  tangent 2-3184
  tangent, inverse 2-237
hyperlink
  displaying in Command Window 2-891
hyperlinks
  in Command Window 2-2035
hyperplanes, angle between 2-3070
hypot 2-1556

i 2-1559
icon images
  reading 2-1622
idealfilter (timeseries) 2-1560
identity matrix 2-1021
  sparse 2-2880
idivide 2-1563
IEEE floating-point arithmetic
  smallest positive number 2-2607
if 2-1565
ifft 2-1569
ifft2 2-1571
ifftn 2-1573
ifftshift 2-1575
IIR filter 2-1174
ilu 2-1576
im2java 2-1581
imag 2-1583
image 2-1584
Image
  creating 2-1584
  properties 2-1591
image types
  querying 2-1610
images
  file formats 2-1620 2-1633
  reading data from files 2-1620
  returning information about 2-1609
  writing to files 2-1633
Images
  converting MATLAB image to Java
    Image 2-1581
imagesc 2-1606
imaginary 2-1583
  part of complex number 2-1583
  unit (sqrt(\xd0 1)) 2-1559 2-1816
  See also complex
imfinfo
  returning file information 2-1609
imformats 2-1612
import 2-1615
importdata 2-1617
importing
  Java class and package names 2-1615
imread 2-1620
imwrite 2-1633
incomplete beta function
  (defined) 2-361
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>incomplete gamma function</td>
<td>2-1343</td>
</tr>
<tr>
<td>(defined)</td>
<td></td>
</tr>
<tr>
<td>ind2sub</td>
<td>2-1648</td>
</tr>
<tr>
<td>Index into matrix is negative or zero (error message)</td>
<td>2-1981</td>
</tr>
<tr>
<td>indexing</td>
<td></td>
</tr>
<tr>
<td>logical</td>
<td>2-1980</td>
</tr>
<tr>
<td>indicator of file position</td>
<td>2-1274</td>
</tr>
<tr>
<td>indices, array</td>
<td></td>
</tr>
<tr>
<td>of sorted elements</td>
<td>2-2855</td>
</tr>
<tr>
<td>Inf</td>
<td>2-1652</td>
</tr>
<tr>
<td>inferiorto</td>
<td>2-1654</td>
</tr>
<tr>
<td>infinity</td>
<td>2-1652</td>
</tr>
<tr>
<td>norm</td>
<td>2-2207</td>
</tr>
<tr>
<td>info</td>
<td>2-1655</td>
</tr>
<tr>
<td>information</td>
<td></td>
</tr>
<tr>
<td>returning file information</td>
<td>2-1609</td>
</tr>
<tr>
<td>inheritance, of objects</td>
<td>2-537</td>
</tr>
<tr>
<td>inline</td>
<td>2-1656</td>
</tr>
<tr>
<td>inmem</td>
<td>2-1659</td>
</tr>
<tr>
<td>inpolygon</td>
<td>2-1661</td>
</tr>
<tr>
<td>input</td>
<td>2-1663</td>
</tr>
<tr>
<td>checking number of M-file arguments</td>
<td>2-2186</td>
</tr>
<tr>
<td>name of array passed as</td>
<td>2-1668</td>
</tr>
<tr>
<td>number of M-file arguments</td>
<td>2-2188</td>
</tr>
<tr>
<td>prompting users for</td>
<td>2-1663</td>
</tr>
<tr>
<td></td>
<td>2-2083</td>
</tr>
<tr>
<td>inputdlg</td>
<td>2-1664</td>
</tr>
<tr>
<td>inputname</td>
<td>2-1668</td>
</tr>
<tr>
<td>inputParser</td>
<td>2-1669</td>
</tr>
<tr>
<td>inspect</td>
<td>2-1675</td>
</tr>
<tr>
<td>installation, root directory of</td>
<td>2-2042</td>
</tr>
<tr>
<td>instrcallback</td>
<td>2-1682</td>
</tr>
<tr>
<td>instrfind</td>
<td>2-1684</td>
</tr>
<tr>
<td>instrfindall</td>
<td>2-1686</td>
</tr>
<tr>
<td>example of</td>
<td>2-1687</td>
</tr>
<tr>
<td>int2str</td>
<td>2-1689</td>
</tr>
<tr>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>floating-point, maximum</td>
<td>2-386</td>
</tr>
<tr>
<td>integration</td>
<td></td>
</tr>
<tr>
<td>polynomial</td>
<td>2-2456</td>
</tr>
<tr>
<td>quadrature</td>
<td>2-2542</td>
</tr>
<tr>
<td>interfaces</td>
<td>2-1692</td>
</tr>
<tr>
<td>interp1</td>
<td>2-1694</td>
</tr>
<tr>
<td>interp1q</td>
<td>2-1702</td>
</tr>
<tr>
<td>interp2</td>
<td>2-1704</td>
</tr>
<tr>
<td>interp3</td>
<td>2-1708</td>
</tr>
<tr>
<td>interpnt</td>
<td>2-1710</td>
</tr>
<tr>
<td>interpolated shading and printing</td>
<td>2-2483</td>
</tr>
<tr>
<td>interpolation</td>
<td></td>
</tr>
<tr>
<td>cubic method</td>
<td>2-1427</td>
</tr>
<tr>
<td>2-1694</td>
<td></td>
</tr>
<tr>
<td>2-1704</td>
<td></td>
</tr>
<tr>
<td>2-1708</td>
<td></td>
</tr>
<tr>
<td>one-dimensional</td>
<td>2-1694</td>
</tr>
<tr>
<td>three-dimensional</td>
<td>2-1708</td>
</tr>
<tr>
<td>trilinear method</td>
<td>2-1427</td>
</tr>
<tr>
<td>two-dimensional</td>
<td>2-1704</td>
</tr>
<tr>
<td>Interpreter</td>
<td></td>
</tr>
<tr>
<td>Text property</td>
<td>2-3213</td>
</tr>
<tr>
<td>textarrow property</td>
<td>2-168</td>
</tr>
<tr>
<td>textbox property</td>
<td>2-180</td>
</tr>
<tr>
<td>interpstreamspeed</td>
<td>2-1714</td>
</tr>
<tr>
<td>Interruptible</td>
<td></td>
</tr>
<tr>
<td>areaseries property</td>
<td>2-204</td>
</tr>
<tr>
<td>Axes property</td>
<td>2-286</td>
</tr>
<tr>
<td>barseries property</td>
<td>2-333</td>
</tr>
<tr>
<td>contour property</td>
<td>2-640</td>
</tr>
<tr>
<td>errorbar property</td>
<td>2-981</td>
</tr>
<tr>
<td>Figure property</td>
<td>2-1116</td>
</tr>
<tr>
<td>hggroup property</td>
<td>2-1515</td>
</tr>
<tr>
<td>hgtransform property</td>
<td>2-1535</td>
</tr>
<tr>
<td>Image property</td>
<td>2-1601</td>
</tr>
<tr>
<td>Light property</td>
<td>2-1896</td>
</tr>
<tr>
<td>Line property</td>
<td>2-1915</td>
</tr>
</tbody>
</table>
lineseries property 2-1928
patch property 2-2352
quivergroup property 2-2568
rectangle property 2-2625
Root property 2-2708
scatter property 2-2768
stairseries property 2-2937
stem property 2-2971
Surface property 2-3111 2-3132
Text property 2-3214
Uicontextmenu property 2-3339
Uicontrol property 2-3360
Uimenu property 2-3401
Uipushtool property 2-3435
Uitoggletool property 2-3466
Uitoolbar property 2-3478
intersect 2-1718
intmax 2-1719
intmin 2-1720
intwarning 2-1721
inv 2-1725
inverse
cosecant 2-76
cosine 2-66
cotangent 2-71
Fourier transform 2-1569 2-1571 2-1573
Hilbert matrix 2-1728
hyperbolic cosecant 2-79
hyperbolic cosine 2-69
hyperbolic cotangent 2-74
hyperbolic secant 2-221
hyperbolic sine 2-226
hyperbolic tangent 2-237
of a matrix 2-1725
secant 2-218
sine 2-223
tangent 2-232
tangent, four-quadrant 2-234
inversion, matrix
accuracy of 2-607
InvertHardCopy, Figure property 2-1117
invhilb 2-1728
invoke 2-1729
involutary matrix 2-2327
ipermute 2-1732
iqr (timeseries) 2-1733
is* 2-1735
isa 2-1737
isappdata function 2-1739
iscell 2-1740
iscellstr 2-1741
ischar 2-1742
iscom 2-1743
isdir 2-1744
isempty 2-1745
isempty (timeseries) 2-1746
isempty (tscollection) 2-1747
isequal 2-1748
isequalwithequalnans 2-1751
isevent 2-1753
isfield 2-1755
isfinite 2-1757
isfloat 2-1758
isglobal 2-1759
ishandle 2-1761
isscalar 2-1763
isinteger 2-1764
isinterface 2-1765
iskeyword 2-1767
islogical 2-1770
ismac 2-1771
ismember 2-1772
ismethod 2-1774
isnan 2-1775
isnumeric 2-1776
isobject 2-1777
isocap 2-1778
isonormals 2-1785
isosurface 2-1788
  calculate data from volume 2-1788
  end caps 2-1778
  vertex normals 2-1785
ispc 2-1792
ispref function 2-1793
isprime 2-1794
isprop 2-1795
isreal 2-1796
isscalar 2-1799
issorted 2-1800
isspace 2-1803 2-1806
issparse 2-1804
isstr 2-1805
isstruct 2-1809
issstudent 2-1810
isunix 2-1811
isvalid 2-1812
  timer object 2-1813
isvarname 2-1814
isvector 2-1815
italics font
  TeX characters 2-3222

J
j 2-1816
  Jacobi rotations 2-2902
  Jacobian elliptic functions
    (defined) 2-947
  Jacobian matrix (BVP) 2-421
  Jacobian matrix (ODE) 2-2262
    generating sparse numerically 2-2263
    2-2265
    specifying 2-2262 2-2265
    vectorizing ODE function 2-2263 to 2-2265
Java
  class names 2-541 2-1615
  objects 2-1766
  Java Image class

creating instance of 2-1581
Java import list
  adding to 2-1615
  clearing 2-541
Java version used by MATLAB 2-3530
java_method 2-1821 2-1828
java_object 2-1830
javaaddath 2-1817
javachk 2-1822
javalargpath 2-1824
jawmpath 2-1832
joining arrays. See concatenation
Joint Photographic Experts Group (JPEG)
  writing 2-1634
JPEG
  setting Bitdepth 2-1638
  specifying mode 2-1638
JPEG comment
  setting when writing a JPEG image 2-1638
JPEG files
  parameters that can be set when
    writing 2-1638
    writing 2-1634
JPEG quality
  setting when writing a JPEG image 2-1638
    2-1643
  setting when writing an HDF image 2-1638
jvm
  version used by MATLAB 2-3530

K
K>> prompt
  keyboard function 2-1836
keyboard 2-1836
keyboard mode 2-1836
  terminating 2-2689
KeyPressFcn
  Uicontrol property 2-3361
KeyPressFcn, Figure property 2-1117
KeyReleaseFcn, Figure property 2-1119
keyword search in functions 2-1989
keywords
  iskeyword function 2-1767
kron 2-1837
Kronecker tensor product 2-1837

L
Label, Uimenu property 2-3402
labeling
  axes 2-3618
  matrix columns 2-891
  plots (with numeric values) 2-2218
LabelSpacing
  contour property 2-640
Laplace 2-829
largest array elements 2-2061
lasterr 2-1839
lasterror 2-1842
lastwarn 2-1846
LaTeX, see TeX 2-170 2-182 2-3220
Layer, Axes property 2-286
Layout Editor
  starting 2-1448
lcm 2-1848
LData
  errorbar property 2-981
LDataSource
  errorbar property 2-981
ldivide (M-file function equivalent for .\) 2-41
le 2-1856
least common multiple 2-1848
least squares
  polynomial curve fitting 2-2452
  problem, overdetermined 2-2413
legend 2-1858
  properties 2-1863
  setting text properties 2-1863
legendre 2-1866
Legendre functions
  (defined) 2-1866
  Schmidt semi-normalized 2-1866
length 2-1870
  serial port I/O 2-1871
length (timeseries) 2-1872
length (tscollection) 2-1873
LevelList
  contour property 2-641
LevelListMode
  contour property 2-641
LevelStep
  contour property 2-641
LevelStepMode
  contour property 2-641
libfunctions 2-1874
libfunctionsview 2-1876
libisloaded 2-1878
libpointer 2-1880
libstruct 2-1882
license 2-1885
light 2-1889
Light
  creating 2-1889
  defining default properties 2-1588 2-1890
  positioning in camera coordinates 2-436
  properties 2-1891
Light object
  positioning in spherical coordinates 2-1899
lightangle 2-1899
lighting 2-1900
limits of axes, setting and querying 2-3620
line 2-1902
  editing 2-2430
Line
  creating 2-1902
  defining default properties 2-1907
  properties 2-1908 2-1921 2-2617
line numbers in M-files 2-783
linear audio signal 2-1901 2-2169
linear dependence (of data) 2-3070
linear equation systems
  accuracy of solution 2-607
  solving overdetermined 2-2532 to 2-2533
linear equation systems, methods for solving
  Cholesky factorization 2-2125
  Gaussian elimination 2-2126
  Householder reflections 2-2127
  matrix inversion (inaccuracy of) 2-1725
linear interpolation 2-1694 2-1704 2-1708 2-1711
linear regression 2-2452
linearly spaced vectors, creating 2-1954
LineColor
  contour property 2-642
lines
  computing 2-D stream 1-101 2-2994
  computing 3-D stream 1-101 2-2996
  drawing stream lines 1-101 2-2998
LineSpec 1-85 2-1937
LineStyle
  annotation arrow property 2-148
  annotation doublearrow property 2-153
  annotation ellipse property 2-157
  annotation line property 2-159
  annotation rectangle property 2-163
  annotation textbox property 2-180
  areaseries property 2-204
  barseries property 2-333
  contour property 2-642
  errorbar property 2-982
  Line property 2-1916
  lineseries property 2-1929
  Patch property 2-2353
  quivergroup property 2-2569
  rectangle property 2-2625
  scatter property 2-2769
  stairseries property 2-2939
  stem property 2-2972
  Surface property 2-3112
  surfaceplot property 2-3134
  text object 2-3216
  textarrow property 2-169
LineStyleOrder
  Axes property 2-286
LineWidth
  annotation arrow property 2-149
  annotation doublearrow property 2-154
  annotation ellipse property 2-157
  annotation line property 2-160
  annotation rectangle property 2-163
  annotation textbox property 2-181
  areaseries property 2-205
  Axes property 2-288
  barseries property 2-334
  contour property 2-643
  errorbar property 2-982
  Line property 2-1916
  lineseries property 2-1929
  Patch property 2-2353
  quivergroup property 2-2569
  rectangle property 2-2625
  scatter property 2-2769
  stairseries property 2-2939
  stem property 2-2972
  Surface property 2-3112
  surfaceplot property 2-3134
  text object 2-3216
  textarrow property 2-169
linkaxes 2-1943
linkprop 2-1947
links
  in Command Window 2-2035
linsolve 2-1951
linspace 2-1954
lint tool for checking problems 2-2129
list boxes 2-3344
  defining items 2-3367
ListboxTop, Uicontrol property 2-3362
listdlg 2-1955
listfonts 2-1958
little endian formats 2-1225
load 2-1960 2-1965
  serial port I/O 2-1966
loadlibrary 2-1968
loadobj 2-1974
Lobatto IIIa ODE solver 2-412
local variables 2-1294 2-1409
locking M-files 2-2139
log 2-1976
  saving session to file 2-880
log10 [log10] 2-1977
log1p 2-1978
log2 2-1979
logarithm
  base ten 2-1977
  base two 2-1979
  complex 2-1976 to 2-1977
  natural 2-1976
  of beta function (natural) 2-363
  of gamma function (natural) 2-1344
  of real numbers 2-2605
  plotting 2-1982
logarithmic derivative
  gamma function 2-2508
logarithmically spaced vectors, creating 2-1988
logical 2-1980
logical array
  converting numeric array to 2-1980
  detecting 2-1770
logical indexing 2-1980
logical operations
  AND, bit-wise 2-382
  OR, bit-wise 2-388
  XOR 2-3645
  XOR, bit-wise 2-392
logical operators 2-48 2-50
logical OR
  bit-wise 2-388
logical tests 2-1737
  all 2-127
  any 2-187

See also detecting
logarithm
  logical XOR 2-3645
    bit-wise 2-392
loglog 2-1982
logm 2-1985
logspace 2-1988
lookfor 2-1989
lossy compression
  writing JPEG files with 2-1638
Lotus WK1 files
  loading 2-3612
  writing 2-3614
lower 2-1991
lower triangular matrix 2-3283
lowercase to uppercase 2-3508
ls 2-1992
lscov 2-1993
lsqnonneg 2-1998
lsqr 2-2001
lt 2-2006
lu 2-2008
LU factorization 2-2008
  storage requirements of (sparse) 2-2222
luinc 2-2016

M
M-file
  debugging 2-1836
  displaying during execution 2-925
  function 2-1294
  function file, echoing 2-925
  naming conventions 2-1294
  pausing execution of 2-2367
  programming 2-1294
  script 2-1294
  script file, echoing 2-925
M-files
  checking existence of 2-1009
  checking for problems 2-2129
clearing from workspace 2-539
creating
    in MATLAB directory 2-2361
debugging with profile 2-2498
deleting 2-848
editing 2-929
line numbers, listing 2-783
lint tool 2-2129
listing names of in a directory 2-3587
locking (preventing clearing) 2-2139
opening 2-2274
optimizing 2-2498
problems, checking for 2-2129
save to HTML 2-2511
setting breakpoints 2-773
unlocking (allowing clearing) 2-2181

M-Lint
    function 2-2129
    function for entire directory 2-2139
    HTML report 2-2135
machine epsilon 2-3596
magic 2-2023
magic squares 2-2023
Margin
    annotation textbox property 2-181
text object 2-3218
Marker
    Line property 2-1916
    lineseries property 2-1930
    Patch property 2-2354
    quivergroup property 2-2570
    scatter property 2-2770
    stairseries property 2-2940
    stem property 2-2973
    Surface property 2-3113
    surfaceplot property 2-3135
MarkerFaceColor
    errorbar property 2-984
    Line property 2-1917
    lineseries property 2-1930
    Patch property 2-2354
    quivergroup property 2-2570
    scatter property 2-2770
    stairseries property 2-2940
    stem property 2-2973
    Surface property 2-3113
    surfaceplot property 2-3135
MarkerSize
    errorbar property 2-984
    Line property 2-1918
    lineseries property 2-1930
    Patch property 2-2355
    quivergroup property 2-2570
    scatter property 2-2770
    stairseries property 2-2940
    stem property 2-2974
    Surface property 2-3114
    surfaceplot property 2-3135
mass matrix (ODE) 2-2266
    initial slope 2-2267 to 2-2268
    singular 2-2267
    sparsity pattern 2-2267
    specifying 2-2267
    state dependence 2-2267
MAT-file 2-2736
    converting sparse matrix after loading
        from 2-2867
MAT-files 2-1960
    listing for directory 2-3587
mat2cell 2-2028
mat2str 2-2031
material 2-2033
MATLAB
  directory location 2-2042
  installation directory 2-2042
  quitting 2-2551
  startup 2-2040
    version number, comparing 2-3528
    version number, displaying 2-3522
matlab : function 2-2035
matlab (UNIX command) 2-2044
matlab (Windows command) 2-2057
matlab function for UNIX 2-2044
matlab function for Windows 2-2057
MATLAB startup file 2-2949
matlab.mat 2-1960 2-2736
matlabcolon function 2-2035
matlabrc 2-2040
matlabroot 2-2042
$matlabroot 2-2042
matrices
  preallocation 2-3648
matrix 2-36
  addressing selected rows and columns of 2-57
  arrowhead 2-592
  companion 2-600
  complex unitary 2-2530
  condition number of 2-607 2-2600
  condition number, improving 2-309
  converting to formatted data file 2-1245
  converting to from string 2-2920
  converting to vector 2-57
  decomposition 2-2530
  defective (defined) 2-933
  detecting sparse 2-1804
  determinant of 2-871
  diagonal of 2-877
  Dulmage-Mendelsohn decomposition 2-908
  evaluating functions of 2-1303
  exponential 2-1016
  flipping left-right 2-1207
  flipping up-down 2-1208
  Hadamard 2-1453 2-3070
  Hankel 2-1454
  Hermitian Toeplitz 2-3273
  Hessenberg form of 2-1500
  Hilbert 2-1540
  identity 2-1021
  inverse 2-1725
  inverse Hilbert 2-1728
  inversion, accuracy of 2-607
  involutary 2-2327
  left division (arithmetic operator) 2-37
  lower triangular 2-3283
  magic squares 2-2023 2-3078
  maximum size of 2-605
  modal 2-931
  multiplication (defined) 2-37
  orthonormal 2-2530
  Pascal 2-2327 2-2459
  permutation 2-2008 2-2530
  poorly conditioned 2-1540
  power (arithmetic operator) 2-38
  pseudoinverse 2-2413
  reading files into 2-900
  reduced row echelon form of 2-2731
  replicating 2-2671
  right division (arithmetic operator) 2-37
  rotating 90\x9f 2-2720
  Schur form of 2-2733 2-2776
  singularity, test for 2-871
  sorting rows of 2-2858
  sparse. See sparse matrix
  specialized 2-1320
  square root of 2-2914
  subspaces of 2-3070
  test 2-1320
  Toeplitz 2-3273
trace of 2-877 2-3275
transpose (arithmetic operator) 2-38
transposing 2-54
unimodular 2-1348
unitary 2-3149
upper triangular 2-3290
Vandermonde 2-2454
Wilkinson 2-2873 2-3607
writing as binary data 2-1308
writing formatted data to 2-1275
writing to ASCII delimited file 2-904
writing to spreadsheet 2-3614
See also array
Matrix
    hgtransform property 2-1536
matrix functions
    evaluating 2-1303
matrix names, (M1 through M12) generating a sequence of 2-997
matrix power. See matrix, exponential
max 2-2061
max (timeseries) 2-2062
Max, Uicontrol property 2-3362
MaxHeadSize
    quivergroup property 2-2571
maximum matching 2-908
MDL-files
    checking existence of 2-1009
mean 2-2066
mean (timeseries) 2-2067
median 2-2069
median (timeseries) 2-2070
median value of array elements 2-2069
memmapfile 2-2076
memory 2-2082
    clearing 2-539
    minimizing use of 2-2308
    variables in 2-3600
menu (of user input choices) 2-2083
menu function 2-2083
MenuBar, Figure property 2-1122
mesh plot
    tetrahedron 2-3189
mesh size (BVP) 2-424
meshc 1-96 2-2085
meshgrid 2-2090
MeshStyle, Surface property 2-3114
MeshStyle, surfaceplot property 2-3136
meshz 1-96 2-2085
message
    error See error message 2-3564
    warning See warning message 2-3564
methods 2-2092
    inheritance of 2-537
    locating 2-3591
methodview 2-2094
mex 2-2096
MEX-files
    clearing from workspace 2-539
    debugging on UNIX 2-764
    listing for directory 2-3587
mexext 2-2098
mfilename 2-2099
mget function 2-2100
Microsoft Excel files
    loading 2-3625
min 2-2101
min (timeseries) 2-2102
Min, Uicontrol property 2-3363
MinColormap, Figure property 2-1122
minimum degree ordering 2-3170
MinorGridLineStyle, Axes property 2-288
minres 2-2106
minus (M-file function equivalent for -) 2-41
mislocked 2-2111
mkdir 2-2112
mkdir (ftp) 2-2115
mkpp 2-2116
mldivide (M-file function equivalent for \) 2-41
mlint 2-2129
mlintrpt 2-2135
    suppressing messages 2-2138
mlock 2-2139
mmfileinfo 2-2140
mod 2-2143
modal matrix 2-931
mode 2-2145
mode objects
    pan, using 2-2312
    rotate3d, using 2-2724
    zoom, using 2-3653
models
    opening 2-2274
    saving 2-2747
modification date
    of a file 2-885
modified Bessel functions
    relationship to Airy functions 2-121
modulo arithmetic 2-2143
MonitorPosition
    Root property 2-2708
Moore-Penrose pseudoinverse 2-2413
more 2-2148 2-2169
move 2-2150
movefile 2-2152
movegui function 2-2155
movie 2-2157
movie2avi 2-2160
movies
    exporting in AVI format 2-252
mpower (M-file function equivalent for ^) 2-42
mput function 2-2162
mdivide (M-file function equivalent for /) 2-41
mtimes 2-2165
mtimes (M-file function equivalent for *) 2-41
mu-law encoded audio signals 2-1901 2-2169
multibandread 2-2170
multibandwrite 2-2175
multidimensional arrays 2-1870
    concatenating 2-459
    interpolation of 2-1711
    longest dimension of 2-1870
    number of dimensions of 2-2197
    rearranging dimensions of 2-1732 2-2405
    removing singleton dimensions of 2-2917
    reshaping 2-2680
    size of 2-2840
    sorting elements of 2-2854
    See also elements
multiple
    least common 2-1848
multiplication
    array (arithmetic operator) 2-37
    matrix (defined) 2-37
    of polynomials 2-656
    multistep ODE solver 2-2242
munlock 2-2181

N
Name, Figure property 2-1123
namelengthmax 2-2183
naming conventions
    M-file 2-1294
NaN 2-2184
NaN (Not-a-Number) 2-2184
    returned by rem 2-2667
nargchk 2-2186
nargoutchk 2-2190
native2unicode 2-2192
ndgrid 2-2195
ndims 2-2197
ne 2-2198
nearest neighbor interpolation 2-1427 2-1694
    2-1704 2-1708 2-1711
newplot 2-2200
NextPlot
    Axes property 2-288
    Figure property 2-1123
nextpow2 2-2203
nnz 2-2204
no derivative method 2-1222
noncontiguous fields, inserting data into 2-1308
nonzero entries
  specifying maximum number of in sparse matrix 2-2864
nonzero entries (in sparse matrix)
  allocated storage for 2-2222
  number of 2-2204
  replacing with ones 2-2894
  vector of 2-2206
nonzeros 2-2206
norm 2-2207
  1-norm 2-2207 2-2600
  2-norm (estimate of) 2-2209
  F-norm 2-2207
  infinity 2-2207
  matrix 2-2207
  pseudoinverse and 2-2413 2-2415
  vector 2-2207
normal vectors, computing for volumes 2-1785
NormalMode
  Patch property 2-2355
  Surface property 2-3114
  surfaceplot property 2-3136
normest 2-2209
not 2-2210
not (M-file function equivalent for ~) 2-49
notebook 2-2211
now 2-2212
nthroot 2-2213
null 2-2214
null space 2-2214
num2cell 2-2216
num2hex 2-2217
num2str 2-2218
number
  of array dimensions 2-2197
numbers
  imaginary 2-1583
  NaN 2-2184
  plus infinity 2-1652
  prime 2-2470
  random 2-2583 2-2588
  real 2-2604
  smallest positive 2-2607
NumberTitle, Figure property 2-1124
numel 2-2220
numeric format 2-1232
numeric precision
  format reading binary data 2-1259
numerical differentiation formula ODE solvers 2-2243
numerical evaluation
  double integral 2-762
  triple integral 2-3285
nzmax 2-2222

object
determining class of 2-1737
inheritance 2-537
object classes, list of predefined 2-536 2-1737
objects
  Java 2-1766
ODE file template 2-2246
ODE solver properties
  error tolerance 2-2253
  event location 2-2260
  Jacobian matrix 2-2262
  mass matrix 2-2266
  ode15s 2-2268
  solver output 2-2255
  step size 2-2259
ODE solvers
  backward differentiation formulas 2-2268
  numerical differentiation formulas 2-2268
  obtaining solutions at specific times 2-2230
variable order solver 2-2268
ode15i function 2-2223
odefile 2-2245
odeget 2-2251
odephas2 output function 2-2257
odephas3 output function 2-2257
odeplot output function 2-2257
odeprint output function 2-2257
odeset 2-2252
odextend 2-2270
off-screen figures, displaying 2-1188
OffCallback
    Uitoggletool property 2-3467
%#ok 2-2130
OnCallback
    Uitoggletool property 2-3468
one-step ODE solver 2-2242
ones 2-2273
online documentation, displaying 2-1494
online help 2-1489
open 2-2274
openfig 2-2278
OpenGL 2-1129
    autoselection criteria 2-1133
opening
    files in Windows applications 2-3608
opening files 2-1225
openvar 2-2285
operating system
    MATLAB is running on 2-605
operating system command 1-4 1-11 2-3179
operating system command, issuing 2-56
operators
    arithmetic 2-36
    logical 2-48 2-50
    overloading arithmetic 2-42
    overloading relational 2-46
    relational 2-46 2-1980
    symbols 2-1489
optimget 2-2287
optimization parameters structure 2-2287 to
    2-2288
optimizing M-file execution 2-2498
optimset 2-2288
or 2-2292
or (M-file function equivalent for |) 2-49
ordeig 2-2294
ordherfd 2-2297
ordering
    minimum degree 2-3170
    reverse Cuthill-McKee 2-3160 2-3171
ordqz 2-2300
ordschur 2-2302
orient 2-2304
orth 2-2306
orthogonal-triangular decomposition 2-2530
orthographic projection, setting and
    querying 2-445
ortnonormal matrix 2-2530
otherwise 2-2307
Out of memory (error message) 2-2308
OuterPosition
    Axes property 2-288
output
    checking number of M-file arguments 2-2190
    controlling display format 2-1232
    in Command Window 2-2148
    number of M-file arguments 2-2188
output points (ODE)
    increasing number of 2-2255
output properties (DDE) 2-807
output properties (ODE) 2-2255
    increasing number of output points 2-2255
overdetermined equation systems,
    solving 2-2532 to 2-2533
overflow 2-1652
overloading
    arithmetic operators 2-42
    relational operators 2-46
    special characters 2-56
P

P-files
  checking existence of 2-1009
pack 2-2308
pagesetupdlg 2-2310
paging
  of screen 2-1491
paging in the Command Window 2-2148
pan mode objects 2-2312
PaperOrientation, Figure property 2-1124
PaperPosition, Figure property 2-1124
PaperPositionMode, Figure property 2-1124
PaperSize, Figure property 2-1125
PaperType, Figure property 2-1125
PaperUnits, Figure property 2-1126
parametric curve, plotting 2-1042
Parent
  areaseries property 2-205
  Axes property 2-290
  barseries property 2-334
  contour property 2-643
  errorbar property 2-984
  Figure property 2-1127
  hggroup property 2-1516
  hgtransform property 2-1536
  Image property 2-1602
  Light property 2-1896
  Line property 2-1918
  lineseries property 2-1931
  Patch property 2-2355
  quivergroup property 2-2571
  rectangle property 2-2626
  Root property 2-2709
  scatter property 2-2770
  stairseries property 2-2940
  stem property 2-2974
  Surface property 2-3114
  surfaceplot property 2-3136
  Text property 2-3219
  Uicontextmenu property 2-3340
Uicontrol property 2-3364
Uimenu property 2-3403
Uipushtool property 2-3436
Uitoggletool property 2-3468
Uitoolbar property 2-3479
parentheses (special characters) 2-54
parse
  inputParser object 2-2321
parseSoapResponse 2-2324
partial fraction expansion 2-2682
partialpath 2-2325
pascal 2-2327
Pascal matrix 2-2327 2-2459
patch 2-2328
Patch
  converting a surface to 1-102 2-3090
  creating 2-2328
  defining default properties 2-2334
  properties 2-2336
  reducing number of faces 1-101 2-2631
  reducing size of face 1-101 2-2829
path 2-2360
  adding directories to 2-107
  building from parts 2-1291
  current 2-2360
  removing directories from 2-2701
  viewing 2-2365
path2rc 2-2362
pathdef 2-2363
pathname
  partial 2-2325
  toolbox directory 1-8 2-3274
pathnames
  of functions or files 2-3591
  relative 2-2325
pathsep 2-2364
pathtool 2-2365
pause 2-2367
pauses, removing 2-757
pausing M-file execution 2-2367
pbaspect 2-2369
PBM
parameters that can be set when writing 2-1638
PBM files
writing 2-1634
pcg 2-2375
pchip 2-2379
pcode 2-2382
pcolor 2-2383
PCX files
writing 2-1635
PDE. See Partial Differential Equations
pdepe 2-2387
pdeval 2-2399
percent sign (special characters) 2-55
percent-brace (special characters) 2-55
perfect matching 2-908
period (.), to distinguish matrix and array operations 2-36
period (special characters) 2-54
perl 2-2402
perl function 2-2402
Perl scripts in MATLAB 1-4 1-11 2-2402
perms 2-2404
permutation
matrix 2-2008 2-2530
of array dimensions 2-2405
random 2-2592
permutations of \(n\) elements 2-2404
permute 2-2405
persistent 2-2406
persistent variable 2-2406
perspective projection, setting and querying 2-445
PGM
parameters that can be set when writing 2-1638
PGM files
writing 2-1635
phase angle, complex 2-142
phase, complex
correcting angles 2-3501
pi 2-2408
pie 2-2409
pie3 2-2411
pinv 2-2413
planerot 2-2416
platform MATLAB is running on 2-605
playshow function 2-2417
plot 2-2418
editing 2-2430
plot (timeseries) 2-2425
plot box aspect ratio of axes 2-2369
plot editing mode
overview 2-2431
Plot Editor
interface 2-2431 2-2505
plot, volumetric
generating grid arrays for 2-2090
slice plot 1-90 1-101 2-2846
PlotBoxAspectRatio, Axes property 2-290
PlotBoxAspectRatioMode, Axes property 2-291
plotedit 2-2430
plotting
2-D plot 2-2418
3-D plot 1-85 2-2426
contours (a 2-1022
contours (ez function) 2-1022
ez-function mesh plot 2-1030
feather plots 2-1062
filled contours 2-1026
function plots 2-1240
functions with discontinuities 2-1050
histogram plots 2-1541
in polar coordinates 2-1045
isosurfaces 2-1788
loglog plot 2-1982
mathematical function 2-1038
mesh contour plot 2-1034
mesh plot 1-96 2-2085
parametric curve 2-1042
plot with two y-axes 2-2437
ribbon plot 1-90 2-2693
rose plot 1-89 2-2716
scatter plot 2-2433
scatter plot, 3-D 1-90 2-2757
semilogarithmic plot 1-86 2-2786
stem plot, 3-D 1-88 2-2960
surface plot 1-96 2-3085
surfaces 1-89 2-1048
velocity vectors 2-611
volumetric slice plot 1-90 1-101 2-2846
. See visualizing
plus (M-file function equivalent for +) 2-41
PNG
writing options for 2-1640
  alpha 2-1640
  background color 2-1640
  chromaticities 2-1641
  gamma 2-1641
  interlace type 2-1641
  resolution 2-1642
  significant bits 2-1641
  transparency 2-1642
PNG files
  writing 2-1635
PNM files
  writing 2-1635
Pointer, Figure property 2-1127
PointerLocation, Root property 2-2709
PointerShapeCData, Figure property 2-1127
PointerShapeHotSpot, Figure property 2-1128
PointerWindow, Root property 2-2710
pol2cart 2-2440
polar 2-2442
polar coordinates 2-2440
  computing the angle 2-142
  converting from Cartesian 2-454
  converting to cylindrical or Cartesian 2-2440
plotting in 2-1045
poles of transfer function 2-2682
poly 2-2444
polyarea 2-2447
polyder 2-2449
polyeig 2-2450
polyfit 2-2452
polygamma function 2-2508
polygon
  area of 2-2447
  creating with patch 2-2328
  detecting points inside 2-1661
polyint 2-2456
polynomial
  analytic integration 2-2456
  characteristic 2-2444 to 2-2445 2-2714
  coefficients (transfer function) 2-2682
  curve fitting with 2-2452
  derivative of 2-2449
  division 2-828
  eigenvalue problem 2-2450
  evaluation 2-2457
  evaluation (matrix sense) 2-2459
  make piecewise 2-2116
  multiplication 2-656
polyval 2-2457
polyvalm 2-2459
poorly conditioned
  matrix 2-1540
  poorly conditioned eigenvalues 2-309
pop-up menus 2-3344
  defining choices 2-3367
Portable Anymap files
  writing 2-1635
Portable Bitmap (PBM) files
  writing 2-1634
Portable Graymap files
  writing 2-1635
Portable Network Graphics files
  writing 2-1635
Portable pixmap format
writing 2-1635
Position
annotation ellipse property 2-157
annotation line property 2-160
annotation rectangle property 2-164
arrow property 2-149
Axes property 2-291
doublearrow property 2-154
Figure property 2-1128
Light property 2-1896
Text property 2-3219
textarrow property 2-170
textbox property 2-181
Uicontrol property 2-3364
Uimenu property 2-3403
position indicator in file 2-1287
position of camera
dollying 2-432
position of camera, setting and querying 2-443
Position, rectangle property 2-2626
PostScript
default printer 2-2475
levels 1 and 2 2-2475
print frame
pow2 2-2461
pow 2-2462
matrix. See matrix exponential
of real numbers 2-2608
of two, next 2-2203
power (M-file function equivalent for .^) 2-42
PPM
parameters that can be set when writing 2-1638
PPM files
writing 2-1635
ppval 2-2463
pragma
%#ok 2-2130
preallocation
matrix 2-3648
precision 2-1232
reading binary data writing 2-1259
prefdir 2-2465
preferences 2-2469
opening the dialog box 2-2469
prime factors 2-1056
dependence of Fourier transform on 2-1076
2-1078 to 2-1079
prime numbers 2-2470
primes 2-2470
print frames 2-1256
printdlg 1-91 1-103 2-2487
printdlg function 2-2487
printer
default for linux and unix 2-2475
printer drivers
GhostScript drivers 2-2472
interpolated shading 2-2483
MATLAB printer drivers 2-2472
printframe 2-1256
PrintFrame Editor 2-1256
printing
borders 2-1256
fig files with frames 2-1256
GUIs 2-2482
interpolated shading 2-2483
on MS-Windows 2-2481
with a variable filename 2-2485
with non-normal EraseMode 2-1913 2-2345
2-2623 2-3107 2-3208
with print frames 2-1258
printing figures
preview 1-92 1-103 2-2488
printing tips 2-2481
printing, suppressing 2-55
printpreview 1-92 1-103 2-2488
prod 2-2496
product
Index

cumulative 2-711
Kronecker tensor 2-1837
of array elements 2-2496
of vectors (cross) 2-698
scalar (dot) 2-698
profile 2-2498
profsave 2-2504
projection type, setting and querying 2-445
ProjectionType, Axes property 2-292
prompting users for input 2-1663 2-2083
propedit 2-2505 to 2-2506
proppanel 1-86 2-2507
pseudoinverse 2-2413
psi 2-2508
publish function 2-2510
push buttons 2-3344
PutFullMatrix 2-2516
pwd 2-2523

Q
qmr 2-2524
qr 2-2530
QR decomposition 2-2530
  deleting column from 2-2535
qrdelete 2-2535
qrinsert 2-2537
qrupdate 2-2539
quad 2-2542
quad1 2-2545
quadrature 2-2542
quadv 2-2547
questdlg 1-103 2-2549
questdlg function 2-2549
quit 2-2551
quitting MATLAB 2-2551
quiver 2-2554
quiver3 2-2557
quotation mark
  inserting in a string 2-1250
qz 2-2580
QZ factorization 2-2451 2-2580

R
radio buttons 2-3344
rand 2-2583
randn 2-2588
random
  numbers 2-2583 2-2588
  permutation 2-2592
  sparse matrix 2-2900 to 2-2901
  symmetric sparse matrix 2-2902
randperm 2-2592
range space 2-2306
rank 2-2593
rank of a matrix 2-2593
RAS files
  parameters that can be set when
  writing 2-1643
  writing 2-1635
RAS image format
  specifying color order 2-1643
  writing alpha data 2-1643
Raster image files
  writing 2-1635
rational fraction approximation 2-2594
rbbox 1-100 2-2598 2-2638
rcond 2-2600
rdivide (M-file function equivalent for ./) 2-41
readasync 2-2601
reading
  binary files 2-1259
  data from files 2-3228
  formatted data from file 2-1275
  formatted data from strings 2-2920
readme files, displaying 1-5 2-1744 2-3590
real 2-2604
real numbers 2-2604
reallog 2-2605

Index-44
realmax 2-2606
realmin 2-2607
realpow 2-2608
realsqrt 2-2609
rearranging arrays
  converting to vector 2-57
  removing first n singleton dimensions 2-2826
  removing singleton dimensions 2-2917
  reshaping 2-2680
  shifting dimensions 2-2826
  swapping dimensions 2-1732 2-2405
rearranging matrices
  converting to vector 2-57
  flipping left-right 2-1207
  flipping up-down 2-1208
  rotating 90\xfb 2-2720
  transposing 2-54
record 2-2610
rectangle
  rectangle function 2-2612
rectint 2-2628
RecursionLimit
  Root property 2-2710
recycle 2-2629
reduced row echelon form 2-2731
reducepatch 2-2631
reducevolume 2-2635
reference page
  accessing from doc 2-910
refresh 2-2638
regexprep 2-2653
regextranslate 2-2657
registerevent 2-2660
regression
  linear 2-2452
  regularly spaced vectors, creating 2-57 2-1954
rehash 2-2663
relational operators 2-46 2-1980
relative accuracy
  BVP 2-420
DDE 2-806
  norm of DDE solution 2-806
  norm of ODE solution 2-2254
  ODE 2-2254
release 2-2665
rem 2-2667
removets 2-2668
rename function 2-2670
renderer
  OpenGL 2-1129
  painters 2-1129
  zbuffer 2-1129
Renderer, Figure property 2-1129
RendererMode, Figure property 2-1133
repeatedly executing statements 2-1230 2-3594
replicating a matrix 2-2671
repmat 2-2671
resample (timeseries) 2-2673
resample (tscollection) 2-2676
reset 2-2679
reshape 2-2680
residue 2-2682
residues of transfer function 2-2682
Resize, Figure property 2-1134
ResizeFcn, Figure property 2-1134
restoredefaultpath 2-2686
rethrow 2-2687
return 2-2689
reverse Cuthill-McKee ordering 2-3160 2-3171
rewinding files to beginning of 2-1274 2-1617
RGB, converting to HSV 1-97 2-2690
rgb2hsv 2-2690
rgbplot 2-2691
ribbon 2-2693
right-click and context menus 2-3332
rmapdata function 2-2695
rmdir 2-2696
rmdir (ftp) function 2-2699
rmfield 2-2700
rmpath 2-2701
rmpref function 2-2702
RMS. See root-mean-square
rolling camera 2-446
root 1-93 2-2703
root directory 2-2042
root directory for MATLAB 2-2042
Root graphics object 1-93 2-2703
root object 2-2703
root, see rootobject 1-93 2-2703
root-mean-square
  of vector 2-2207
roots 2-2714
roots of a polynomial 2-2444 to 2-2445 2-2714
rose 2-2716
Rosenbrock
  banana function 2-1220
  ODE solver 2-2243
rosser 2-2719
rot90 2-2720
rotate 2-2721
rotate3d 2-2724
rotate3d mode objects 2-2724
rotating camera 2-440
rotating camera target 1-98 2-442
Rotation, Text property 2-3219
rotations
  Jacobi 2-2902
round 2-2730
  to nearest integer 2-2730
  towards infinity 2-484
  towards minus infinity 2-1210
  towards zero 2-1205
roundoff error
  characteristic polynomial and 2-2445
  convolution theorem and 2-656
  effect on eigenvalues 2-309
  evaluating matrix functions 2-1305
  in inverse Hilbert matrix 2-1728
  partial fraction expansion and 2-2683
  polynomial roots and 2-2714
  sparse matrix conversion and 2-2868
rref 2-2731
rrefmovie 2-2731
rsf2csf 2-2733
rubberband box 1-100 2-2598
run 2-2735
Runge-Kutta ODE solvers 2-2242
running average 2-1175
S
  save 2-2736 2-2744
    serial port I/O 2-2745
  saveas 2-2747
  saveobj 2-2751
  savepath 2-2753
  saving
    ASCII data 2-2736
    session to a file 2-880
    workspace variables 2-2736
  scalar product (of vectors) 2-698
  scaled complementary error function (defined) 2-965
  scatter 2-2754
  scatter3 2-2757
  scattered data, aligning
    multi-dimensional 2-2195
    two-dimensional 2-1427
  scattergroup
    properties 2-2760
Schmidt semi-normalized Legendre functions 2-1866
schur 2-2776
  Schur decomposition 2-2776
  Schur form of matrix 2-2733 2-2776
screen, paging 2-1491
ScreenDepth, Root property 2-2710
ScreenPixelsPerInch, Root property 2-2711
ScreenSize, Root property 2-2711
script 2-2779
scrolling screen 2-1491
search path 2-2701
adding directories to 2-107
MATLAB’s 2-2360
 modifying 2-2365
viewing 2-2365
search, string 2-1192
sec 2-2780
secant 2-2780
hyperbolic 2-2783
inverse 2-218
inverse hyperbolic 2-221
secd 2-2782
sech 2-2783
Selected
areaseres property 2-205
Axes property 2-292
barseres property 2-335
contour property 2-643
errobar property 2-984
Figure property 2-1136
hgggroup property 2-1516
hgttransform property 2-1536
Image property 2-1602
Light property 2-1897
Line property 2-1918
lineseries property 2-1931
Patch property 2-2356
quivergroup property 2-2571
rectangle property 2-2626
scatter property 2-2771
stairseries property 2-2941
stem property 2-2974
Surface property 2-3115
surfaceplot property 2-3137
Text property 2-3220
Uicontrol property 2-3365
SelectionType, Figure property 2-1136
selectmoversize 2-2785
semicolon (special characters) 2-55
send 2-2789
sendmail 2-2790
Separator
Uipushtool property 2-3436
Uitoggletool property 2-3468
Separator, Uimenu property 2-3403
sequence of matrix names (M1 through M12)
generating 2-997
serial 2-2792
serialbreak 2-2794
server (FTP)
 connecting to 2-1288
server variable 2-1068
session
 saving 2-880
set 1-112 2-2795 2-2799
serial port I/O 2-2800
timer object 2-2803
set (timeseries) 2-2806
set (tscollection) 2-2807
set operations
difference 2-2811
exclusive or 2-2823
intersection 2-1718
membership 2-1772
union 2-3483
unique 2-3485
setabstime (timeseries) 2-2808
setabstime (tscollection) 2-2809
setappdata 2-2810
setdiff 2-2811
setenv 2-2812
setfield 2-2813
setinterpmethod 2-2815
setpixelposition 2-2817
setpref function 2-2820
setstr 2-2821
settimesteriablesnames 2-2822
setxor 2-2823
shading 2-2824
shading colors in surface plots 1-97 2-2824
ShareColors, Figure property 2-1137
shared libraries
MATLAB functions
calllib 2-429
libfunctions 2-1874
libfunctionsview 2-1876
libisloaded 2-1878
libpointer 2-1880
libstruct 2-1882
loadlibrary 2-1968
unloadlibrary 2-3490
shell script 1-4 1-11 2-3179 2-3488
shiftdim 2-2826
shifting array
circular 2-528
ShowArrowHead
quivergroup property 2-2572
ShowBaseLine
barseries property 2-335
ShowHiddenHandles, Root property 2-2712
showplottool 2-2827
ShowText
contour property 2-643
shrinkfaces 2-2829
shutdown 2-2551
sign 2-2833
signum function 2-2833
simplex search 2-1222
Simpson's rule, adaptive recursive 2-2543
Simulink
printing diagram with frames 2-1256
version number, comparing 2-3528
version number, displaying 2-3522
sin 2-2834
sind 2-2836
sine 2-2834
hyperbolic 2-2838
inverse 2-223
inverse hyperbolic 2-226
single 2-2837
single quote (special characters) 2-54
singular value
decomposition 2-2593 2-3149
largest 2-2207
rank and 2-2593
sinh 2-2838
size
array dimensions 2-2840
serial port I/O 2-2843
size (timeseries) 2-2844
size (tscollection) 2-2845
size of array dimensions 2-2840
size of fonts, see also FontSize property 2-3222
size vector 2-2680
Index

SizeData
   scatter property 2-2771
skipping bytes (during file I/O) 2-1308
slice 2-2846
slice planes, contouring 2-651
sliders 2-3345
SliderStep, Uicontrol property 2-3365
smallest array elements 2-2101
smooth3 2-2852
smoothing 3-D data 1-101 2-2852
soccer ball (example) 2-3171
solution statistics (BVP) 2-425
sort 2-2854
sorting
   array elements 2-2854
   complex conjugate pairs 2-691
   matrix rows 2-2858
sortrows 2-2858
sound 2-2861 to 2-2862
   converting vector into 2-2861 to 2-2862 files
      reading 2-250 2-3575
      writing 2-251 2-3580
   playing 1-81 2-3573
   recording 1-82 2-3578
   resampling 1-81 2-3573
   sampling 1-82 2-3578
source control on UNIX platforms
   checking out files
      function 2-510
source control system
   viewing current system 2-553
source control systems
   checking in files 2-507
   undo checkout 1-10 2-3481
spalloc 2-2863
sparse 2-2864
sparse matrix
   allocating space for 2-2863
   applying function only to nonzero elements of 2-2881
density of 2-2204
detecting 2-1804
diagonal 2-2869
finding indices of nonzero elements of 2-1182
identity 2-2880
minimum degree ordering of 2-559
number of nonzero elements in 2-2204
permuting columns of 2-592
random 2-2900 to 2-2901
random symmetric 2-2902
replacing nonzero elements of with ones 2-2894
results of mixed operations on 2-2865
solving least squares linear system 2-2531
specifying maximum number of nonzero elements 2-2864
vector of nonzero elements 2-2206
visualizing sparsity pattern of 2-2911
sparse storage
   criterion for using 2-1290
spaugment 2-2866
spconvert 2-2867
spdiags 2-2869
special characters
   descriptions 2-1489
   overloading 2-56
specular 2-2879
SpecularColorReflectance
   Patch property 2-2356
   Surface property 2-3115
   surfaceplot property 2-3137
SpecularExponent
   Patch property 2-2356
   Surface property 2-3115
   surfaceplot property 2-3137
SpecularStrength
   Patch property 2-2356
   Surface property 2-3115
surfaceplot property 2-3137
speye 2-2880
spfun 2-2881
sph2cart 2-2883
sphere 2-2884
spherical coordinates
  defining a Light position in 2-1899
spherical coordinates 2-2883
spinmap 2-2886
spline 2-2887
spline interpolation (cubic)
  one-dimensional 2-1695 2-1705 2-1708
  2-1711
Spline Toolbox 2-1700
spones 2-2894
spparms 2-2895
sprand 2-2900
sprandn 2-2901
sprandsym 2-2902
sprank 2-2903
spreadsheets
  loading WK1 files 2-3612
  loading XLS files 2-3625
  reading into a matrix 2-900
  writing from matrix 2-3614
  writing matrices into 2-904
sprintf 2-2904
sqrt 2-2913
sqrtm 2-2914
square root
  of a matrix 2-2914
  of array elements 2-2913
  of real numbers 2-2609
squeeze 2-2917
sscanf 2-2920
stack, displaying 2-767
standard deviation 2-2950
start
  timer object 2-2946
startat
timer object 2-2947
startup 2-2949
startup file 2-2949
startup files 2-2040
State
  Uitoggletool property 2-3468
Stateflow
  printing diagram with frames 2-1256
static text 2-3345
std 2-2950
std (timeseries) 2-2952
stem 2-2954
stem3 2-2960
step size (DDE)
  initial step size 2-810
  upper bound 2-811
step size (ODE) 2-809 2-2259
  initial step size 2-2259
  upper bound 2-2259
stop
  timer object 2-2980
stopasync 2-2981
stopwatch timer 2-3255
storage
  allocated for nonzero entries (sparse) 2-2222
  sparse 2-2864
storage allocation 2-3648
str2cell 2-500
str2double 2-2983
str2func 2-2984
str2mat 2-2986
str2num 2-2987
strcat 2-2989
stream lines
  computing 2-D 1-101 2-2994
  computing 3-D 1-101 2-2996
  drawing 1-101 2-2998
stream2 2-2994
stream3 2-2996
stretch-to-fill 2-260
strfind 2-3026
string
  comparing one to another 2-2991 2-3032
  converting from vector to 2-506
  converting matrix into 2-2031 2-2218
  converting to lowercase 2-1991
  converting to numeric array 2-2987
  converting to uppercase 2-3508
  dictionary sort of 2-2858
  finding first token in 2-3043
  searching and replacing 2-3042
  searching for 2-1192
String
  Text property 2-3220
  textarrow property 2-170
  textbox property 2-181
  Uicontrol property 2-3366
string matrix to cell array conversion 2-500
strings 2-3028
  converting to matrix (formatted) 2-2920
  inserting a quotation mark in 2-1250
  writing data to 2-2904
strjust 1-52 1-64 2-3030
strmatch 2-3031
strread 2-3034
strrep 1-52 1-64 2-3042
strtok 2-3043
strtrim 2-3046
struct 2-3047
struct2cell 2-3052
structfun 2-3053
structure array
  getting contents of field of 2-1380
  remove field from 2-2700
  setting contents of a field of 2-2813
structure arrays
  field names of 2-1096
structures
  dynamic fields 2-55
strvcat 2-3056
Style
  Light property 2-1897
    Uicontrol property 2-3368
sub2ind 2-3058
subfunction 2-1294
subplot 2-3060
subsasgn 1-55 2-3067
subscripts
    in axis title 2-3271
    in text strings 2-3224
subsindex 2-3069
subspace 1-20 2-3070
subsref 1-55 2-3071
subsref (M-file function equivalent for
    A(i,j,k...)) 2-56
substruct 2-3073
subtraction (arithmetic operator) 2-36
subvolume 2-3075
sum 2-3078
    cumulative 2-713
    of array elements 2-3078
sum (timeseries) 2-3081
superiorto 2-3083
superscripts
    in axis title 2-3271
    in text strings 2-3224
support 2-3084
surf2patch 2-3090
surface 2-3092
Surface
    and contour plotter 2-1052
    converting to a patch 1-102 2-3090
    creating 1-93 1-96 2-3092
    defining default properties 2-2616 2-3096
    plotting mathematical functions 2-1048
    properties 2-3097 2-3118
    surface normals, computing for volumes 2-1785
surf1 2-3143
surf2norm 2-3147
svd 2-3149
svds 2-3152
swapbytes 2-3155
switch 2-3157
symamd 2-3159
symbfact 2-3163
symbols
  operators 2-1489
symbols in text 2-170 2-182 2-3220
symmlq 2-3165
symmmd 2-3170
symrcm 2-3171
synchronize 2-3174
syntax 2-1490
syntax, command 2-3176
syntax, function 2-3176
syntaxes
  of M-file functions, defining 2-1294
system 2-3179
  UNC pathname error 2-3179
system directory, temporary 2-3187

T
table lookup. See interpolation
Tag
  areaseres property 2-206
  Axes property 2-292
  barseries property 2-335
  contour property 2-644
  errorbar property 2-985
  Figure property 2-1137
  hggroup property 2-1516
  hgtransform property 2-1537
  Image property 2-1602
  Light property 2-1897
  Line property 2-1919
  lineseries property 2-1931
  Patch property 2-2357
  quivergroup property 2-2572
  rectangle property 2-2626
  Root property 2-2712
  scatter property 2-2771
  stairseries property 2-2941
  stem property 2-2975
  Surface property 2-3116
  surfaceplot property 2-3138
  Text property 2-3225
  Uicontextmenu property 2-3340
  Uicontrol property 2-3368
  Uimenu property 2-3404
  Uipushtool property 2-3436
  Uitoggletool property 2-3469
  Uitoolbar property 2-3479
Tagged Image File Format (TIFF)
  writing 2-1636
tan 2-3181
tand 2-3183
tangent 2-3181
  four-quadrant, inverse 2-234
  hyperbolic 2-3184
  inverse 2-232
  inverse hyperbolic 2-237
tanh 2-3184
tar 2-3186
target, of camera 2-447
tcip 2-3510
tempdir 2-3187
tempname 2-3188
temporary
  files 2-3188
    system directory 2-3187
tensor, Kronecker product 2-1837
terminating MATLAB 2-2551
test matrices 2-1320
test, logical. See logical tests and detecting
tetrahedron
  mesh plot 2-3189
tetramesh 2-3189
TeX commands in text 2-170 2-182 2-3220
text 2-3194
Index

editing 2-2430
subscripts 2-3224
superscripts 2-3224

Text
creating 1-93 2-3194
defining default properties 2-3198
fixed-width font 2-3209
properties 2-3199
text mode for opened files 2-1224

TextBackgroundColor
textarrow property 2-172

TextColor
textarrow property 2-172

TextEdgeColor
textarrow property 2-172

TextLineWidth
textarrow property 2-173

TextList
contour property 2-644

TextListMode
contour property 2-645

TextMargin

textarrow property 2-173
textread 1-77 2-3228

TextRotation, textarrow property 2-173
textscan 1-77 2-3234

TextStep
contour property 2-645

TextStepMode
contour property 2-646

textwrap 2-3254

TickDir, Axes property 2-293
TickDirMode, Axes property 2-293

TickLength, Axes property 2-293

TIFF
compression 2-1643
encoding 2-1639
ImageDescription field 2-1643
maxvalue 2-1639

parameters that can be set when
writing 2-1643
resolution 2-1644
writemode 2-1644
writing 2-1636

TIFF image format
specifying compression 2-1643
tiling (copies of a matrix) 2-2671
time
CPU 2-692
elapsed (stopwatch timer) 2-3255
required to execute commands 2-993
time and date functions 2-960
timer
properties 2-3256
timer object 2-3256
timerfind
timer object 2-3263
timerfindall
timer object 2-3265
times (M-file function equivalent for .*) 2-41
timeseries 2-3267
timestamp 2-885
title 2-3270
with superscript 2-3271

Title, Axes property 2-294
todatenum 2-3272
toeplitz 2-3273
Toeplitz matrix 2-3273
toggle buttons 2-3345
token 2-3043
See also string

Toolbar
Figure property 2-1138

Toolbox
Spline 2-1700
toolbox directory, pathname 1-8 2-3274
toolboxdir 2-3274
TooltipString
Uicontrol property 2-3369
Index

Uipushtool property 2-3437
Uitoggletool property 2-3469
trace 2-3275
trace of a matrix 2-877 2-3275
trailing blanks
removing 2-820
transform
  hgtransform function 2-1523
transform, Fourier
  discrete, n-dimensional 2-1079
  discrete, one-dimensional 2-1073
  discrete, two-dimensional 2-1078
  inverse, n-dimensional 2-1573
  inverse, one-dimensional 2-1569
  inverse, two-dimensional 2-1571
  shifting the zero-frequency component of 2-1082
transformation
  See also conversion 2-470
transformations
  elementary Hermite 2-1348
transmitting file to FTP server 1-84 2-2162
transpose
  array (arithmetic operator) 2-38
  matrix (arithmetic operator) 2-38
  transpose (M-file function equivalent for \q) 2-42
  transpose (timeseries) 2-3276
trapz 2-3278
treelayout 2-3280
treeplot 2-3281
triangulation
  2-D plot 2-3287
tricubic interpolation 2-1427
tril 2-3283
trilinear interpolation 2-1427
trimesh 2-3284
triple integral
  numerical evaluation 2-3285
triplequad 2-3285
triplot 2-3287
trisurf 2-3289
triu 2-3290
true 2-3291
truth tables (for logical operations) 2-48
try 2-3292
tscollecti...
Uitoggletool property 2-3469
Uitoolbar property 2-3479
typecast 2-3307

U
UData
   errorbar property 2-986
   quivergroup property 2-2573
UDataSource
   errorbar property 2-986
   quivergroup property 2-2573
Uibuttongroup
   defining default properties 2-3315
uibuttongroup function 2-3311
Uibuttongroup Properties 2-3315
uicontextmenu 2-3332
UiContextMenu
   Uicontrol property 2-3369
UIContextMenu
   areaseries property 2-207
   Axes property 2-295
   barseries property 2-336
   contour property 2-646
   errorbar property 2-986
   Figure property 2-1139
   hggroup property 2-1517
   hgtransform property 2-1537
   Image property 2-1603
   Light property 2-1898
   Line property 2-1919
   lineseries property 2-1932
   Patch property 2-2357
   quivergroup property 2-2573
   rectangle property 2-2627
   scatter property 2-2772
   stairseries property 2-2942
   stem property 2-2975
   Surface property 2-3116
   surfaceplot property 2-3138
   Text property 2-3226
   Uicontextmenu Properties 2-3334
   uicontrol 2-3342
   Uicontrol
      defining default properties 2-3348
      fixed-width font 2-3357
      types of 2-3342
   Uicontrol Properties 2-3348
   uigetdir 2-3372
   uigetfile 2-3377
   uigetpref function 2-3387
   uimport 2-3391
   umenu 2-3392
   Uimenu
      creating 1-106 2-3392
      defining default properties 2-3394
      Properties 2-3394
   Uimenu Properties 2-3394
   uint16 2-3405
   uint32 2-3405
   uint64 2-3405
   uint8 2-1690 2-3405
   uiopepen 2-3407
   Uipanel
      defining default properties 2-3411
   uipanel function 2-3409
   Uipanel Properties 2-3411
   uipushtool 2-3427
   Uipushtool
      defining default properties 2-3429
   Uipushtool Properties 2-3429
   uiputfile 2-3439
   uiresume 2-3448
   uisave 2-3450
   uisetcolor function 2-3453
   uisefont 2-3454
   uisetpref function 2-3456
   uistack 2-3457
   uitoggletool 2-3458
   Uitoggletool

Index-55
Index

- defining default properties 2-3460
- Uitoggletool Properties 2-3460
- uitoolbar 2-3471
- Uitoolbar
  - defining default properties 2-3473
- Uitoolbar Properties 2-3473
- uwait 2-3448
- uminus (M-file function equivalent for unary \(\times 0\)) 2-41
- UNC pathname error and dos 2-916
- UNC pathname error and system 2-3179
- unconstrained minimization 2-1218
- undefined numerical results 2-2184
- undocheckout 2-3481
- unicode2native 2-3482
- unimodular matrix 2-1348
- union 2-3483
- unique 2-3485
- unitary matrix (complex) 2-2530
- Units
  - annotation ellipse property 2-158
  - annotation rectangle property 2-164
  - arrow property 2-149
  - Axes property 2-295
  - doublearrow property 2-154
  - Figure property 2-1139
  - line property 2-160
  - Root property 2-2713
  - Text property 2-3225
  - textarrow property 2-173
  - textbox property 2-184
  - Uicontrol property 2-3369
- unix 2-3488
- UNIX
  - Web browser 2-912
- unloadlibrary 2-3490
- unlocking M-files 2-2181
- unmkpp 2-3491
- unregisterallevents 2-3492
- unregisterevent 2-3495
- untar 2-3499
- unwrap 2-3501
- unzip 2-3506
- up vector, of camera 2-449
- updating figure during M-file execution 2-921
- uplus (M-file function equivalent for unary +) 2-41
- upper 2-3508
- upper triangular matrix 2-3290
- uppercase to lowercase 2-1991
- url
  - opening in Web browser 1-5 1-8 2-3581
- urlread 2-3509
- urlwrite 2-3511
- usejava 2-3513
- UserData
  - areasersies property 2-207
  - Axes property 2-296
  - barseries property 2-336
  - contour property 2-646
  - errorbar property 2-987
  - Figure property 2-1140
  - hggroup property 2-1517
  - hgtransform property 2-1538
  - Image property 2-1603
  - Light property 2-1898
  - Line property 2-1919
  - lineseries property 2-1932
  - Patch property 2-2358
  - quivergroup property 2-2573
  - rectangle property 2-2627
  - Root property 2-2713
  - scatter property 2-2772
  - stairseries property 2-2942
  - stem property 2-2976
  - Surface property 2-3116
  - surfaceplot property 2-3139
  - Text property 2-3226
  - Uicontextmenu property 2-3341
  - Uicontrol property 2-3370
Uimenu property 2-3404
Uipushtool property 2-3437
Uitoggletool property 2-3469
Uitoobar property 2-3480

V
Value, Uicontrol property 2-3370
vander 2-3515
Vandermonde matrix 2-2454
var 2-3516
var (timeseries) 2-3517
varargin 2-3519
varargout 2-3520
variable numbers of M-file arguments 2-3520
variable-order solver (ODE) 2-2268
variables
  checking existence of 2-1009
  clearing from workspace 2-539
  global 2-1409
  graphical representation of 2-3616
  in workspace 2-3616
  listing 2-3600
  local 2-1294 2-1409
  name of passed 2-1668
  opening 2-2274 2-2285
  persistent 2-2406
  saving 2-2736
  sizes of 2-3600
VData
  quivergroup property 2-2574
VDataSource
  quivergroup property 2-2574
vector
  dot product 2-917
  frequency 2-1988
  length of 2-1870
  product (cross) 2-698
vector field, plotting 2-611
vectorize 2-3521
vectorizing ODE function (BVP) 2-421
vectors, creating
  logarithmically spaced 2-1988
  regularly spaced 2-57 2-1954
velocity vectors, plotting 2-611
ver 2-3522
verctrl function (Windows) 2-3524
verLessThan 2-3528
version 2-3530
version numbers
  comparing 2-3528
  displaying 2-3522
vertcat 2-3532
vertcat (M-file function equivalent for [ 2-56
vertcat (timeseries) 2-3534
vertcat (tscollection) 2-3535
VertexNormals
  Patch property 2-2358
  Surface property 2-3117
  surfaceplot property 2-3139
VerticalAlignment, Text property 2-3226
VerticalAlignment, textarrow property 2-174
VerticalAlignment, textbox property 2-184
Vertices, Patch property 2-2358
video
  saving in AVI format 2-252
view 2-3536
  azimuth of viewpoint 2-3537
  coordinate system defining 2-3537
  elevation of viewpoint 2-3537
view angle, of camera 2-451
View, Axes property (obsolete) 2-296
viewing
  a group of object 2-438
  a specific object in a scene 2-438
viewmtx 2-3539
Visible
  areaseries property 2-207
  Axes property 2-296
  barseries property 2-336
contour property 2-646
errorbar property 2-987
Figure property 2-1140
hggroup property 2-1518
hgtransform property 2-1538
Image property 2-1604
Light property 2-1898
Line property 2-1919
lineseries property 2-1933
Patch property 2-2358
quivergroup property 2-2573
rectangle property 2-2627
Root property 2-2713
scatter property 2-2772
stairseries property 2-2942
stem property 2-2976
Surface property 2-3117
surfaceplot property 2-3139
Text property 2-3227
Uicontextmenu property 2-3341
Uicontrol property 2-3371
Uimenu property 2-3404
Uipushtool property 2-3437
Uitoggletool property 2-3470
Uitoolbar property 2-3480
visualizing
  cell array structure 2-498
  sparse matrices 2-2911
volumes
  calculating isosurface data 2-1788
  computing 2-D stream lines 1-101 2-2994
  computing 3-D stream lines 1-101 2-2996
  computing isosurface normals 2-1785
  contouring slice planes 2-651
  drawing stream lines 1-101 2-2998
  end caps 2-1778
  reducing face size in isosurfaces 1-101
    2-2829
  reducing number of elements in 1-101 2-2635
  voronoi 2-3546
Voronoi diagrams
  multidimensional visualization 2-3552
  two-dimensional visualization 2-3546
voronoin 2-3552

W
  wait
    timer object 2-3556
  waitbar 2-3557
  waitfor 2-3559
  waitforbuttonpress 2-3560
  waitforbuttonpress 2-3560
  warndlg 2-3561
  warning 2-3564
  warning message (enabling, suppressing, and
displaying) 2-3564
  waterfall 2-3568
  .wav files
    reading 2-3575
    writing 2-3580
  waverecord 2-3578
  wavfinfo 2-3572
  wavplay 1-81 2-3573
  wavread 2-3572 2-3575
  wavrecord 1-82 2-3578
  wavwrite 2-3580
WData
  quivergroup property 2-2575
WDataSource
  quivergroup property 2-2575
web 2-3581
Web browser
  displaying help in 2-1494
    pointing to file or url 1-5 1-8 2-3581
    specifying for UNIX 2-912
weekday 2-3585
well conditioned 2-2600
what 2-3587
whatsnew 2-3590
which 2-3591
while 2-3594
white space characters, ASCII 2-1803 2-3043
whitebg 2-3598
who, whos
who 2-3600
wilkinson 2-3607
Wilkinson matrix 2-2873 2-3607
WindowButtonDownFcn, Figure property 2-1140
WindowButtonMotionFcn, Figure property 2-1141
WindowButtonDownFcn, Figure property 2-1141
Windows Paintbrush files
writing 2-1635
WindowScrollWheelFcn, Figure property 2-1142
WindowStyle, Figure property 2-1145
winopen 2-3608
winqueryreg 2-3609
WK1 files
loading 2-3612
writing from matrix 2-3614
wk1finfo 2-3611
wk1read 2-3612
wk1write 2-3614
workspace 2-3616
changing context while debugging 2-761 2-784
clearing items from 2-539
consolidating memory 2-2308
predefining variables 2-2949
saving 2-2736
variables in 2-3600
viewing contents of 2-3616
workspace variables
reading from disk 2-1960
writing
binary data to file 2-1308
formatted data to file 2-1245
WVisual, Figure property 2-1147
WVisualMode, Figure property 2-1149

X
x
annotation arrow property 2-150 2-154
annotation line property 2-161
texarrow property 2-175
X Windows Dump files
writing 2-1636
x-axis limits, setting and querying 2-3620
XAxisLocation, Axes property 2-296
XColor, Axes property 2-297
XData
areaseries property 2-207
barseries property 2-337
contour property 2-647
errorbar property 2-987
Image property 2-1604
Line property 2-1920
lineseries property 2-1933
Patch property 2-2358
quivergroup property 2-2576
scatter property 2-2773
stairseries property 2-2943
stem property 2-2976
Surface property 2-3117
surfaceplot property 2-3139
XDataMode
areaseries property 2-208
barseries property 2-337
contour property 2-647
errorbar property 2-987
lineseries property 2-1933
quivergroup property 2-2576
stairseries property 2-2943
stem property 2-2976
surfaceplot property 2-3139
XDataSource
areaseries property 2-208
barseries property 2-337
contour property 2-647
errorbar property 2-988
lineseries property 2-1934
quivergroup property 2-2577
scatter property 2-2773
stairseries property 2-2943
stem property 2-2977
surfaceplot property 2-3140
XDir, Axes property 2-297
XDisplay, Figure property 2-1149
XGrid, Axes property 2-298
xlabel 1-87 2-3618
XLabel, Axes property 2-298
xlim 2-3620
XLim, Axes property 2-299
XLimMode, Axes property 2-299
XLS files
    loading 2-3625
xlsfinfo 2-3623
xlsread 2-3625
xlswrite 2-3635
XMinorGrid, Axes property 2-300
xmlread 2-3639
xmlwrite 2-3644
xor 2-3645
XOR, printing 2-201 2-330 2-637 2-978 2-1533
    2-1599 2-1913 2-1925 2-2346 2-2565 2-2623
    2-2765 2-2935 2-2968 2-3107 2-3128 2-3208
XScale, Axes property 2-300
xslt 2-3646
XTick, Axes property 2-300
XTickLabel, Axes property 2-301
XTickLabelMode, Axes property 2-302
XTickMode, Axes property 2-302
XVisual, Figure property 2-1150
XVisualMode, Figure property 2-1152
XWD files
    writing 2-1636
xyz coordinates. See Cartesian coordinates

Y
Y
    annotation arrow property 2-150 2-155 2-161
textarrow property 2-175
y-axis limits, setting and querying 2-3620
YAxisLocation, Axes property 2-297
YColor, Axes property 2-297
YData
    areasersies property 2-209
    barseries property 2-338
    contour property 2-648
    errorbar property 2-988
    Image property 2-1604
    Line property 2-1920
    lineseries property 2-1934
    Patch property 2-2359
    quivergroup property 2-2577
    scatter property 2-2774
    stairseries property 2-2944
    stem property 2-2977
    Surface property 2-3117
    surfaceplot property 2-3140
YDataMode
    contour property 2-649
    quivergroup property 2-2578
    surfaceplot property 2-3141
YDataSource
    areasersies property 2-209
    barseries property 2-338
    contour property 2-649
    errorbar property 2-989
    lineseries property 2-1934
    quivergroup property 2-2578
    scatter property 2-2774
    stairseries property 2-2944
    stem property 2-2978
    surfaceplot property 2-3141
YDir, Axes property 2-297
YGrid, Axes property 2-298
ylabel 1-87 2-3618
YLabel, Axes property 2-298
ylim 2-3620
YLim, Axes property 2-299
YLimMode, Axes property 2-299
YMinorGrid, Axes property 2-300
YScale, Axes property 2-300
YTick, Axes property 2-300
YTickLabel, Axes property 2-301
YTickLabelMode, Axes property 2-302
YTickMode, Axes property 2-302
surfaceplot property 2-3142
ZDataSource
  contour property 2-650
  lineseries property 2-1935 2-2979
  scatter property 2-2775
  surfaceplot property 2-3142
ZDir, Axes property 2-297
zero of a function, finding 2-1314
zeros 2-3648
ZGrid, Axes property 2-298
zip 2-3650
zlabel 1-87 2-3618
zlim 2-3620
ZLim, Axes property 2-299
ZLimMode, Axes property 2-299
ZMinorGrid, Axes property 2-300
zoom 2-3652
zoom mode objects 2-3653
ZScale, Axes property 2-300
ZTick, Axes property 2-300
ZTickLabel, Axes property 2-301
ZTickLabelMode, Axes property 2-302
ZTickMode, Axes property 2-302

Z
z-axis limits, setting and querying 2-3620
ZColor, Axes property 2-297
ZData
  contour property 2-649
  Line property 2-1920
  lineseries property 2-1935
  Patch property 2-2359
  quivergroup property 2-2579
  scatter property 2-2774
  stemseries property 2-2978
  Surface property 2-3117