Contents

Functions — Categorical List

1

Desktop Tools and Development Environment ........ 1-4
  Startup and Shutdown .............................. 1-4
  Command Window and History ..................... 1-5
  Help for Using MATLAB ............................ 1-5
  Workspace, Search Path, and File Operations .......... 1-5
  Programming Tools ................................ 1-7
  System ........................................... 1-8

Mathematics ....................................... 1-9
  Arrays and Matrices ................................ 1-10
  Linear Algebra .................................... 1-12
  Elementary Math ................................... 1-14
  Data Analysis and Fourier Transforms .............. 1-17
  Polynomials ....................................... 1-18
  Interpolation and Computational Geometry .......... 1-19
  Coordinate System Conversion ..................... 1-20
  Nonlinear Numerical Methods ....................... 1-20
  Specialized Math .................................. 1-22
  Sparse Matrices ................................... 1-22
  Math Constants .................................... 1-24

Programming and Data Types ......................... 1-25
  Data Types ........................................ 1-25
  Arrays ............................................ 1-30
  Operators and Operations .......................... 1-32
  Programming in MATLAB ............................. 1-35

File I/O ........................................... 1-40
  Filename Construction .............................. 1-40
  Opening, Loading, Saving Files .................... 1-41
  Low-Level File I/O ................................ 1-41
  Text Files ........................................ 1-41
  XML Documents .................................... 1-41
  Spreadsheets ...................................... 1-42
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scientific Data</td>
<td>1-42</td>
</tr>
<tr>
<td>Audio and Audio/Video</td>
<td>1-43</td>
</tr>
<tr>
<td>Images</td>
<td>1-43</td>
</tr>
<tr>
<td>Internet Exchange</td>
<td>1-44</td>
</tr>
<tr>
<td><strong>Graphics</strong></td>
<td>1-45</td>
</tr>
<tr>
<td>Basic Plots and Graphs</td>
<td>1-45</td>
</tr>
<tr>
<td>Annotating Plots</td>
<td>1-46</td>
</tr>
<tr>
<td>Specialized Plotting</td>
<td>1-46</td>
</tr>
<tr>
<td>Bit-Mapped Images</td>
<td>1-49</td>
</tr>
<tr>
<td>Printing</td>
<td>1-49</td>
</tr>
<tr>
<td>Handle Graphics</td>
<td>1-49</td>
</tr>
<tr>
<td><strong>3-D Visualization</strong></td>
<td>1-52</td>
</tr>
<tr>
<td>Surface and Mesh Plots</td>
<td>1-52</td>
</tr>
<tr>
<td>View Control</td>
<td>1-53</td>
</tr>
<tr>
<td>Lighting</td>
<td>1-55</td>
</tr>
<tr>
<td>Transparency</td>
<td>1-55</td>
</tr>
<tr>
<td>Volume Visualization</td>
<td>1-55</td>
</tr>
<tr>
<td><strong>Creating Graphical User Interfaces</strong></td>
<td>1-56</td>
</tr>
<tr>
<td>Predefined Dialog Boxes</td>
<td>1-56</td>
</tr>
<tr>
<td>Deploying User Interfaces</td>
<td>1-57</td>
</tr>
<tr>
<td>Developing User Interfaces</td>
<td>1-57</td>
</tr>
<tr>
<td>User Interface Objects</td>
<td>1-57</td>
</tr>
<tr>
<td>Finding Objects from Callbacks</td>
<td>1-57</td>
</tr>
</tbody>
</table>
Functions — Categorical List

The MATLAB® Function Reference contains descriptions of all MATLAB commands and functions.

Select a category from the following table to see a list of related functions.

<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, data analysis, other areas of mathematics</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>Creating Graphical User Interface</td>
<td>GUIDE, programming graphical user interfaces.</td>
</tr>
<tr>
<td>External Interfaces</td>
<td>Java, COM, Serial Port functions.</td>
</tr>
</tbody>
</table>

See Simulink®, Stateflow®, Real-Time Workshop®, and the individual toolboxes for lists of their functions.
**Desktop Tools and Development Environment**

General functions for working in MATLAB, including functions for startup, Command Window, help, and editing and debugging.

“Startup and Shutdown”  Startup and shutdown options
“Command Window and History”  Controlling Command Window and History
“Help for Using MATLAB”  Finding information
“Workspace, Search Path, and File Operations”  File, search path, variable management
“Programming Tools”  Editing and debugging, source control, Notebook
“System”  Identifying current computer, license, product version, and more

**Startup and Shutdown**

- `exit`  Terminate MATLAB (same as `quit`)
- `finish`  MATLAB termination M-file
- `genpath`  Generate a path string
- `matlab`  Start MATLAB (UNIX systems)
- `matlab`  Start MATLAB (Windows systems)
- `matlabrc`  MATLAB startup M-file for single user systems or administrators
- `prefdir`  Return directory containing preferences, history, and layout files
- `preferences`  Display 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 the Command History, or select it if already open
- `commandwindow` Open the Command Window, or select it if already open
- `diary` Save session to file
- `dos` Execute DOS command and return result
- `format` Control display format for output
- `home` Move cursor to upper left corner of Command Window
- `matlab:` Run specified function via hyperlink (matlabcolon)
- `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**

- `doc` Display online documentation in MATLAB Help browser
- `demo` Access product demos via Help browser
- `docopt` Web browser for UNIX platforms
- `docsearch` Open Help browser Search pane and run search for specified term
- `help` Display help for MATLAB functions in Command Window
- `helpbrowser` Display Help browser for access to full online documentation and demos
- `helpwin` Provide access to and display M-file help for all functions
- `info` Display Release Notes for MathWorks products
- `lookfor` Search for specified keyword in all help entries
- `playshow` Run published M-file demo
- `support` Open MathWorks Technical Support Web page
- `web` Open Web site or file in Web browser or Help browser
- `whatsnew` Display Release Notes for MathWorks products

**Workspace, Search Path, and File Operations**

- “Workspace”
- “Search Path”
- “File Operations”
### Workspace

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignin</td>
<td>Assign value to workspace variable</td>
</tr>
<tr>
<td>clear</td>
<td>Remove items from workspace, freeing up system memory</td>
</tr>
<tr>
<td>evalin</td>
<td>Execute string containing MATLAB expression in a workspace</td>
</tr>
<tr>
<td>exist</td>
<td>Check if variables or functions are defined</td>
</tr>
<tr>
<td>openvar</td>
<td>Open workspace variable in Array Editor for graphical editing</td>
</tr>
<tr>
<td>pack</td>
<td>Consolidate workspace memory</td>
</tr>
<tr>
<td>uimport</td>
<td>Open Import Wizard, the graphical user interface to import data</td>
</tr>
<tr>
<td>which</td>
<td>Locate functions and files</td>
</tr>
<tr>
<td>who, whos</td>
<td>List variables in the workspace</td>
</tr>
<tr>
<td>workspace</td>
<td>Display Workspace browser, a tool for managing the workspace</td>
</tr>
</tbody>
</table>

### Search Path

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addpath</td>
<td>Add directories to MATLAB search path</td>
</tr>
<tr>
<td>genpath</td>
<td>Generate path string</td>
</tr>
<tr>
<td>partialpath</td>
<td>Partial pathname</td>
</tr>
<tr>
<td>path</td>
<td>View or change the MATLAB directory search path</td>
</tr>
<tr>
<td>path2rc</td>
<td>Replaced by savepath</td>
</tr>
<tr>
<td>pathdef</td>
<td>List of directories in the MATLAB search path</td>
</tr>
<tr>
<td>pathsep</td>
<td>Return path separator for current platform</td>
</tr>
<tr>
<td>pathdef</td>
<td>Open Set Path dialog box to view and change MATLAB 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 pathdef.m file</td>
</tr>
</tbody>
</table>

### File Operations

<table>
<thead>
<tr>
<th>Command</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>Delete files or graphics objects</td>
</tr>
<tr>
<td>dir</td>
<td>Display directory listing</td>
</tr>
<tr>
<td>exist</td>
<td>Check if variables or functions are defined</td>
</tr>
<tr>
<td>fileattrib</td>
<td>Set or get attributes of file or directory</td>
</tr>
<tr>
<td>filebrowser</td>
<td>Display Current Directory browser, a tool for viewing files</td>
</tr>
<tr>
<td>lookfor</td>
<td>Search for specified keyword in all help entries</td>
</tr>
<tr>
<td>ls</td>
<td>List directory on UNIX</td>
</tr>
<tr>
<td>matlabroot</td>
<td>Return root directory of MATLAB installation</td>
</tr>
<tr>
<td>mkdir</td>
<td>Make new directory</td>
</tr>
<tr>
<td>movefile</td>
<td>Move file or directory</td>
</tr>
<tr>
<td>pwd</td>
<td>Display current directory</td>
</tr>
<tr>
<td>recycle</td>
<td>Set option to move deleted files to recycle folder</td>
</tr>
<tr>
<td>rehash</td>
<td>Refresh function and file system path caches</td>
</tr>
<tr>
<td>rmdir</td>
<td>Remove directory</td>
</tr>
</tbody>
</table>
Desktop Tools and Development Environment

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>List file</td>
</tr>
<tr>
<td>web</td>
<td>Open Web site or file in Web browser or Help browser</td>
</tr>
<tr>
<td>what</td>
<td>List MATLAB specific files in current directory</td>
</tr>
<tr>
<td>which</td>
<td>Locate functions and files</td>
</tr>
</tbody>
</table>

See also “File I/O” functions.

**Programming Tools**

- “Editing and Debugging”
- “Performance Improvement and Tuning Tools and Techniques”
- “Source Control”
- “Publishing”

**Editing and Debugging**

dbclear    Clear breakpoints
dbcont     Resume execution
dbdown     Change local workspace context
dbquit     Quit debug mode
dbstack    Display function call stack
dbstatus   List all breakpoints
dbstep     Execute one or more lines from current breakpoint
dbstop     Set breakpoints
dbtype     List M-file with line numbers
dbup       Change local workspace context
debug      M-file debugging functions
edit       Edit or create M-file
keyboard    Invoke the keyboard in an M-file

**Performance Improvement and Tuning Tools and Techniques**

memory     Help for memory limitations
mlint      Check M-files for possible problems, and report results
mlintrpt   Run mlint for file or directory, reporting results in Web browser
pack       Consolidate workspace memory
profile    Profile the execution time for a function
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 file into source control system
- **checkout** Check file out of source control system
- **cmopts** Get name of source control system
- **customverctrl** Allow custom source control system
- **undocheckout** Undo previous checkout from source control system
- **verctrl** Version control operations on PC platforms

### Publishing

- **notebook** Open M-book in Microsoft Word (Windows only)
- **publish** Run M-file containing cells, and save results to file of specified type

### System

- **computer** Identify information about computer on which MATLAB is running
- **javachk** Generate error message based on Java feature support
- **license** Show license number for MATLAB
- **prefdir** Return directory containing preferences, history, and layout files
- **usejava** Determine if a Java feature is supported in MATLAB
- **ver** Display version information for MathWorks products
- **version** Get MATLAB version number
Mathematics

Functions for working with arrays and matrices, linear algebra, data analysis, and other areas of mathematics.

“Arrays and Matrices” Basic array operators and operations, creation of elementary and specialized arrays and matrices

“Linear Algebra” Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization

“Elementary Math” Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math

“Data Analysis and Fourier Transforms” Descriptive statistics, finite differences, correlation, filtering and convolution, fourier transforms

“Polynomials” Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion

“Interpolation and Computational Geometry” Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation

“Coordinate System Conversion” Conversions between Cartesian and polar or spherical coordinates

“Nonlinear Numerical Methods” Differential equations, optimization, integration

“Specialized Math” Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions

“Sparse Matrices” Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations

“Math Constants” Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy
Arrays and Matrices

- “Basic Information”
- “Operators”
- “Operations and Manipulation”
- “Elementary Matrices and Arrays”
- “Specialized Matrices”

Basic Information

disp     Display array
display  Display array
isempty  True for empty matrix
isequal  True if arrays are identical
isfloat  True for floating-point arrays
isinteger True for integer arrays
islogical True for logical array
isnumeric True for numeric arrays
isscalar True for scalars
issparse True for sparse matrix
isvector True for vectors
length   Length of vector
ndims    Number of dimensions
numel    Number of elements
size     Size of matrix

Operators

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

: (colon)  Index into array, rearrange array
accumarray  Construct an array with accumulation
blkdiag  Block diagonal concatenation
cat  Concatenate arrays
cross  Vector cross product
cumprod  Cumulative product
cumsum  Cumulative sum
diag  Diagonal matrices and diagonals of matrix
dot  Vector dot product
dim  Last index
find  Find indices of nonzero elements
fliplr  Flip matrices left-right
flipud  Flip matrices up-down
flipdim  Flip matrix along specified dimension
horzcat  Horizontal concatenation
ind2sub  Multiple subscripts from linear index
ipermute  Inverse permute dimensions of multidimensional array
kron  Kronecker tensor product
max  Maximum value of array
min  Minimum value of array
permute  Rearrange dimensions of multidimensional array
prod  Product of array elements
repmat  Replicate and tile array
reshape  Reshape array
rot90  Rotate matrix 90 degrees
sort  Sort array elements in ascending or descending order
sortrows  Sort rows in ascending order
sum  Sum of array elements
sqrtm  Matrix square root
sub2ind  Linear index from multiple subscripts
tril  Lower triangular part of matrix
triu  Upper triangular part of matrix
vertcat  Vertical concatenation

See also “Linear Algebra” for other matrix operations.
See also “Elementary Math” for other array operations.
Elementary Matrices and Arrays

- `:` (colon) Regularly spaced vector
- `blkdiag` Construct block diagonal matrix from input arguments
- `diag` Diagonal matrices and diagonals of matrix
- `eye` Identity matrix
- `freqspace` Frequency spacing for frequency response
- `linspace` Generate linearly spaced vectors
- `logspace` Generate logarithmically spaced vectors
- `meshgrid` Generate X and Y matrices for three-dimensional plots
- `ndgrid` Arrays for multidimensional functions and interpolation
- `ones` Create array of all ones
- `rand` Uniformly distributed random numbers and arrays
- `randn` Normally distributed random numbers and arrays
- `repmat` Replicate and tile array
- `zeros` Create array of all zeros

Specialized Matrices

- `compan` Companion matrix
- `gallery` Test matrices
- `hadamard` Hadamard matrix
- `hankel` Hankel matrix
- `hilb` Hilbert matrix
- `invhilb` Inverse of Hilbert matrix
- `magic` Magic square
- `pascal` Pascal matrix
- `rosser` Classic symmetric eigenvalue test problem
- `toeplitz` Toeplitz matrix
- `vander` Vandermonde matrix
- `wilkinson` Wilkinson’s eigenvalue test matrix

Linear Algebra

- “Matrix Analysis”
- “Linear Equations”
- “Eigenvalues and Singular Values”
- “Matrix Logarithms and Exponentials”
- “Factorization”
**Matrix Analysis**

- **cond**: Condition number with respect to inversion
- **condeig**: Condition number with respect to eigenvalues
- **det**: Determinant
- **norm**: Matrix or vector norm
- **normest**: Estimate matrix 2-norm
- **null**: Null space
- **orth**: Orthogonalization
- **rank**: Matrix rank
- **rcond**: Matrix reciprocal condition number estimate
- **rref**: Reduced row echelon form
- **subspace**: Angle between two subspaces
- **trace**: Sum of diagonal elements

**Linear Equations**

- **\ and /**: Linear equation solution
- **chol**: Cholesky factorization
- **cholinc**: Incomplete Cholesky factorization
- **cond**: Condition number with respect to inversion
- **condest**: 1-norm condition number estimate
- **funm**: Evaluate general matrix function
- **inv**: Matrix inverse
- **linsolve**: Solve linear systems of equations
- **lscov**: Least squares solution in presence of known covariance
- **lsqnonneg**: Nonnegative least squares
- **lu**: LU matrix factorization
- **luinc**: Incomplete LU factorization
- **pinv**: Moore-Penrose pseudoinverse of matrix
- **qr**: Orthogonal-triangular decomposition
- **rcond**: Matrix reciprocal condition number estimate

**Eigenvalues and Singular Values**

- **balance**: Improve accuracy of computed eigenvalues
- **cdf2rdf**: Convert complex diagonal form to real block diagonal form
- **condeig**: Condition number with respect to eigenvalues
- **eig**: Eigenvalues and eigenvectors
- **eigs**: Eigenvalues and eigenvectors of sparse matrix
- **gsvd**: Generalized singular value decomposition
- **hess**: Hessenberg form of matrix
- **poly**: Polynomial with specified roots
- **polyeig**: Polynomial eigenvalue problem
- **qz**: QZ factorization for generalized eigenvalues
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsf2csf</td>
<td>Convert real Schur form to complex Schur form</td>
</tr>
<tr>
<td>schur</td>
<td>Schur decomposition</td>
</tr>
<tr>
<td>svd</td>
<td>Singular value decomposition</td>
</tr>
<tr>
<td>svds</td>
<td>Singular values and vectors of sparse matrix</td>
</tr>
</tbody>
</table>

**Matrix Logarithms and Exponentials**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expm</td>
<td>Matrix exponential</td>
</tr>
<tr>
<td>logm</td>
<td>Matrix logarithm</td>
</tr>
<tr>
<td>sqrtm</td>
<td>Matrix square root</td>
</tr>
</tbody>
</table>

**Factorization**

<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>Complex diagonal form to real block diagonal form</td>
</tr>
<tr>
<td>chol</td>
<td>Cholesky factorization</td>
</tr>
<tr>
<td>cholinc</td>
<td>Incomplete Cholesky factorization</td>
</tr>
<tr>
<td>cholupdate</td>
<td>Rank 1 update to Cholesky factorization</td>
</tr>
<tr>
<td>lu</td>
<td>LU matrix factorization</td>
</tr>
<tr>
<td>luinc</td>
<td>Incomplete LU factorization</td>
</tr>
<tr>
<td>planerot</td>
<td>Givens plane rotation</td>
</tr>
<tr>
<td>qr</td>
<td>Orthogonal-triangular decomposition</td>
</tr>
<tr>
<td>qrdelete</td>
<td>Delete column or row from QR factorization</td>
</tr>
<tr>
<td>qrinsert</td>
<td>Insert column or row into QR factorization</td>
</tr>
<tr>
<td>qrupdate</td>
<td>Rank 1 update to QR factorization</td>
</tr>
<tr>
<td>qz</td>
<td>QZ factorization for generalized eigenvalues</td>
</tr>
<tr>
<td>rsf2csf</td>
<td>Real block diagonal form to complex diagonal form</td>
</tr>
</tbody>
</table>

**Elementary Math**

- “Trigonometric”
- “Exponential”
- “Complex”
- “Rounding and Remainder”
- “Discrete Math (e.g., Prime Factors)”
### Trigonometric

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>Inverse cosine</td>
</tr>
<tr>
<td>acosd</td>
<td>Inverse cosine, degrees</td>
</tr>
<tr>
<td>acosh</td>
<td>Inverse hyperbolic cosine</td>
</tr>
<tr>
<td>acot</td>
<td>Inverse cotangent</td>
</tr>
<tr>
<td>acotd</td>
<td>Inverse cotangent, degrees</td>
</tr>
<tr>
<td>acoth</td>
<td>Inverse hyperbolic cotangent</td>
</tr>
<tr>
<td>acsc</td>
<td>Inverse cosecant</td>
</tr>
<tr>
<td>acscd</td>
<td>Inverse cosecant, degrees</td>
</tr>
<tr>
<td>acsch</td>
<td>Inverse hyperbolic cosecant</td>
</tr>
<tr>
<td>asec</td>
<td>Inverse secant</td>
</tr>
<tr>
<td>asecd</td>
<td>Inverse secant, degrees</td>
</tr>
<tr>
<td>asech</td>
<td>Inverse hyperbolic secant</td>
</tr>
<tr>
<td>asin</td>
<td>Inverse sine</td>
</tr>
<tr>
<td>asind</td>
<td>Inverse sine, degrees</td>
</tr>
<tr>
<td>asinh</td>
<td>Inverse hyperbolic sine</td>
</tr>
<tr>
<td>atan</td>
<td>Inverse tangent</td>
</tr>
<tr>
<td>atand</td>
<td>Inverse tangent, degrees</td>
</tr>
<tr>
<td>atanh</td>
<td>Inverse hyperbolic tangent</td>
</tr>
<tr>
<td>atan2</td>
<td>Four-quadrant inverse tangent</td>
</tr>
<tr>
<td>cos</td>
<td>Cosine</td>
</tr>
<tr>
<td>cosd</td>
<td>Cosine, degrees</td>
</tr>
<tr>
<td>cosh</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>cot</td>
<td>Cotangent</td>
</tr>
<tr>
<td>cotd</td>
<td>Cotangent, degrees</td>
</tr>
<tr>
<td>coth</td>
<td>Hyperbolic cotangent</td>
</tr>
<tr>
<td>csc</td>
<td>Cosecant</td>
</tr>
<tr>
<td>cscd</td>
<td>Cosecant, degrees</td>
</tr>
<tr>
<td>csch</td>
<td>Hyperbolic cosecant</td>
</tr>
<tr>
<td>sec</td>
<td>Secant</td>
</tr>
<tr>
<td>secd</td>
<td>Secant, degrees</td>
</tr>
<tr>
<td>sech</td>
<td>Hyperbolic secant</td>
</tr>
<tr>
<td>sin</td>
<td>Sine</td>
</tr>
<tr>
<td>sind</td>
<td>Sine, degrees</td>
</tr>
<tr>
<td>sinh</td>
<td>Hyperbolic sine</td>
</tr>
<tr>
<td>tan</td>
<td>Tangent</td>
</tr>
<tr>
<td>tand</td>
<td>Tangent, degrees</td>
</tr>
<tr>
<td>tanh</td>
<td>Hyperbolic tangent</td>
</tr>
</tbody>
</table>
Functions — Categorical List

### Exponential

- **exp**: Exponential
- **expm1**: Exponential of x minus 1
- **log**: Natural logarithm
- **log1p**: Logarithm of 1+x
- **log2**: Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
- **log10**: Common (base 10) logarithm
- **nextpow2**: Next higher power of 2
- **pow2**: Base 2 power and scale floating-point number
- **reallog**: Natural logarithm for nonnegative real arrays
- **realpow**: Array power for real-only output
- **realsqrt**: Square root for nonnegative real arrays
- **sqrt**: Square root
- **nthroot**: Real nth root

### Complex

- **abs**: Absolute value
- **angle**: Phase angle
- **complex**: Construct complex data from real and imaginary parts
- **conj**: Complex conjugate
- **cplxpair**: Sort numbers into complex conjugate pairs
- **i**: Imaginary unit
- **imag**: Complex imaginary part
- **isreal**: True for real array
- **j**: Imaginary unit
- **real**: Complex real part
- **sign**: Signum
- **unwrap**: Unwrap phase angle

### Rounding and Remainder

- **fix**: Round towards zero
- **floor**: Round towards minus infinity
- **ceil**: Round towards plus infinity
- **round**: Round towards nearest integer
- **mod**: Modulus after division
- **rem**: Remainder after division
Discrete Math (e.g., Prime Factors)

- `factor` Prime factors
- `factorial` Factorial function
- `gcd` Greatest common divisor
- `isprime` True for prime numbers
- `lcm` Least common multiple
- `nchoosek` All combinations of N elements taken K at a time
- `perms` All possible permutations
- `primes` Generate list of prime numbers
- `rat`, `rats` Rational fraction approximation

Data Analysis and Fourier Transforms

- “Basic Operations”
- “Finite Differences”
- “Correlation”
- “Filtering and Convolution”
- “Fourier Transforms”

Basic Operations

- `cumprod` Cumulative product
- `cumsum` Cumulative sum
- `cumtrapz` Cumulative trapezoidal numerical integration
- `max` Maximum elements of array
- `mean` Average or mean value of arrays
- `median` Median value of arrays
- `min` Minimum elements of array
- `prod` Product of array elements
- `sort` Sort array elements in ascending or descending order
- `sortrows` Sort rows in ascending order
- `std` Standard deviation
- `sum` Sum of array elements
- `trapz` Trapezoidal numerical integration
- `var` Variance

Finite Differences

- `del2` Discrete Laplacian
- `diff` Differences and approximate derivatives
- `gradient` Numerical gradient
## Correlation
- **corrcosf**: Correlation coefficients
- **cov**: Covariance matrix
- **subspace**: Angle between two subspaces

## Filtering and Convolution
- **conv**: Convolution and polynomial multiplication
- **conv2**: Two-dimensional convolution
- **convn**: N-dimensional convolution
- **deconv**: Deconvolution and polynomial division
- **detrend**: Linear trend removal
- **filter**: Filter data with infinite impulse response (IIR) or finite impulse response (FIR) filter
- **filter2**: Two-dimensional digital filtering

## Fourier Transforms
- **abs**: Absolute value and complex magnitude
- **angle**: Phase angle
- **fft**: One-dimensional discrete Fourier transform
- **fftn**: N-dimensional discrete Fourier Transform
- **fftw**: Interface to the FFTW library run-time algorithm for tuning FFTs
- **ifft**: Inverse one-dimensional discrete Fourier transform
- **ifft2**: Inverse two-dimensional discrete Fourier transform
- **ifftn**: Inverse multidimensional discrete Fourier transform
- **ifftshift**: Inverse fast Fourier transform shift
- **nextpow2**: Next power of two
- **unwrap**: Correct phase angles

## 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**: Analytic polynomial integration
- **polyval**: Polynomial evaluation
- **polyvalm**: Matrix polynomial evaluation
- **residue**: Convert between partial fraction expansion and polynomial coefficients
- **roots**: Polynomial roots
Interpolation and Computational Geometry

- “Interpolation”
- “Delaunay Triangulation and Tessellation”
- “Convex Hull”
- “Voronoi Diagrams”
- “Domain Generation”

Interpolation

- dsearch: Search for nearest point
- dsearchn: Multidimensional closest point search
- griddata: Data gridding
- griddata3: Data gridding and hypersurface fitting for three-dimensional data
- griddatan: Data gridding and hypersurface fitting (dimension \(\geq 2\))
- interp1: One-dimensional data interpolation (table lookup)
- interp2: Two-dimensional data interpolation (table lookup)
- interp3: Three-dimensional data interpolation (table lookup)
- interpft: One-dimensional interpolation using fast Fourier transform method
- interpn: Multidimensional data interpolation (table lookup)
- meshgrid: Generate X and Y matrices for three-dimensional plots
- mkpp: Make piecewise polynomial
- ndgrid: Generate arrays for multidimensional functions and interpolation
- pchip: Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
- ppval: Piecewise polynomial evaluation
- spline: Cubic spline data interpolation
- tsearchn: Multidimensional closest simplex search
- unmkpp: Piecewise polynomial details

Delaunay Triangulation and Tessellation

- delaunay: Delaunay triangulation
- delaunay3: Three-dimensional Delaunay tessellation
- delaunayn: Multidimensional Delaunay tessellation
- dsearch: Search for nearest point
- dsearchn: Multidimensional closest point search
- tetramesh: Tetrahedron mesh plot
- trimesh: Triangular mesh plot
- triplot: Two-dimensional triangular plot
- trisurf: Triangular surface plot
- tsearch: Search for enclosing Delaunay triangle
- tsearchn: Multidimensional closest simplex search
Convex Hull
convhull  Convex hull
convhulln Multidimensional convex hull
patch   Create patch graphics object
plot    Linear two-dimensional plot
trisurf Triangular surface plot

Voronoi Diagrams
dsearch  Search for nearest point
patch    Create patch graphics object
plot     Linear two-dimensional plot
voronoi  Voronoi diagram
voronoin Multidimensional Voronoi diagrams

Domain Generation
meshgrid Generate X and Y matrices for three-dimensional plots
ndgrid   Generate arrays for multidimensional functions and interpolation

Coordinate System Conversion

Cartesian
cart2sph  Transform Cartesian to spherical coordinates
cart2pol  Transform Cartesian to polar coordinates
pol2cart  Transform polar to Cartesian coordinates
sph2cart  Transform spherical to Cartesian coordinates

Nonlinear Numerical Methods
• “Ordinary Differential Equations (IVP)”
• “Delay Differential Equations”
• “Boundary Value Problems”
• “Partial Differential Equations”
• “Optimization”
• “Numerical Integration (Quadrature)”
### Ordinary Differential Equations (IVP)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ode113</td>
<td>Solve non-stiff differential equations, variable order method</td>
</tr>
<tr>
<td>ode15i</td>
<td>Solve fully implicit differential equations, variable order method</td>
</tr>
<tr>
<td>ode15s</td>
<td>Solve stiff ODEs and DAEs Index 1, variable order method</td>
</tr>
<tr>
<td>ode23</td>
<td>Solve non-stiff differential equations, low order method</td>
</tr>
<tr>
<td>ode23s</td>
<td>Solve stiff differential equations, low order method</td>
</tr>
<tr>
<td>ode23t</td>
<td>Solve moderately stiff ODEs and DAEs Index 1, trapezoidal rule</td>
</tr>
<tr>
<td>ode23tb</td>
<td>Solve stiff differential equations, low order method</td>
</tr>
<tr>
<td>ode45</td>
<td>Solve non-stiff differential equations, medium order method</td>
</tr>
<tr>
<td>odextend</td>
<td>Extend the solution of an initial value problem</td>
</tr>
<tr>
<td>odeget</td>
<td>Get ODE options parameters</td>
</tr>
<tr>
<td>odeset</td>
<td>Create/alter ODE options structure</td>
</tr>
<tr>
<td>decic</td>
<td>Compute consistent initial conditions for ode15i</td>
</tr>
<tr>
<td>deval</td>
<td>Evaluate solution of differential equation problem</td>
</tr>
</tbody>
</table>

### Delay Differential Equations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dde23</td>
<td>Solve delay differential equations with constant delays</td>
</tr>
<tr>
<td>ddeget</td>
<td>Get DDE options parameters</td>
</tr>
<tr>
<td>ddeset</td>
<td>Create/alter DDE options structure</td>
</tr>
<tr>
<td>deval</td>
<td>Evaluate solution of differential equation problem</td>
</tr>
</tbody>
</table>

### Boundary Value Problems

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bvp4c</td>
<td>Solve boundary value problems for ODEs</td>
</tr>
<tr>
<td>bvpget</td>
<td>Get BVP options parameters</td>
</tr>
<tr>
<td>bvpset</td>
<td>Create/alter BVP options structure</td>
</tr>
<tr>
<td>deval</td>
<td>Evaluate solution of differential equation problem</td>
</tr>
</tbody>
</table>

### Partial Differential Equations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdepe</td>
<td>Solve initial-boundary value problems for parabolic-elliptic PDEs</td>
</tr>
<tr>
<td>pdeval</td>
<td>Evaluates by interpolation solution computed by pdepe</td>
</tr>
</tbody>
</table>

### Optimization

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fminbnd</td>
<td>Scalar bounded nonlinear function minimization</td>
</tr>
<tr>
<td>fminsearch</td>
<td>Multidimensional unconstrained nonlinear minimization, by Nelder-Mead direct search method</td>
</tr>
<tr>
<td>fzero</td>
<td>Scalar nonlinear zero finding</td>
</tr>
<tr>
<td>lsqnonneg</td>
<td>Linear least squares with nonnegativity constraints</td>
</tr>
<tr>
<td>optimset</td>
<td>Create or alter optimization options structure</td>
</tr>
<tr>
<td>optimget</td>
<td>Get optimization parameters from options structure</td>
</tr>
</tbody>
</table>
### Numerical Integration (Quadrature)
- **quad**  Numerically evaluate integral, adaptive Simpson quadrature (low order)
- **quadl**  Numerically evaluate integral, adaptive Lobatto quadrature (high order)
- **quadv**  Vectorized quadrature
- **dblquad**  Numerically evaluate double integral
- **triplequad**  Numerically evaluate triple integral

### Specialized Math
- **airy**  Airy functions
- **besselh**  Bessel functions of third kind (Hankel functions)
- **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**  Error function
- **erfc**  Complementary error function
- **erfcinv**  Inverse complementary error function
- **erfcx**  Scaled complementary error function
- **erfinv**  Inverse error function
- **expint**  Exponential integral
- **gamma**  Gamma function
- **gammainc**  Incomplete gamma function
- **gammaln**  Logarithm of gamma function
- **legendre**  Associated Legendre functions
- **psi**  Psi (polygamma) function

### Sparse Matrices
- “Elementary Sparse Matrices”
- “Full to Sparse Conversion”
- “Working with Sparse Matrices”
- “Reordering Algorithms”
- “Linear Algebra”
- “Linear Equations (Iterative Methods)”
- “Tree Operations”
Elementary Sparse Matrices

- `spdiags` - Sparse matrix formed from diagonals
- `spye` - Sparse identity matrix
- `sprand` - Sparse uniformly distributed random matrix
- `sprandn` - Sparse normally distributed random matrix
- `sprandsym` - Sparse random symmetric matrix

Full to Sparse Conversion

- `find` - Find indices of nonzero elements
- `full` - Convert sparse matrix to full matrix
- `sparse` - Create sparse matrix
- `spconvert` - Import from sparse matrix external format

Working with Sparse Matrices

- `issparse` - True for sparse matrix
- `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 matrix elements
- `spones` - Replace nonzero sparse matrix elements with ones
- `spparms` - Set parameters for sparse matrix routines
- `spy` - Visualize sparsity pattern

Reordering Algorithms

- `colamd` - Column approximate minimum degree permutation
- `colmmd` - Column minimum degree permutation
- `colperm` - Column permutation
- `dmperm` - Dulmage-Mendelsohn permutation
- `randperm` - Random permutation
- `symamd` - Symmetric approximate minimum degree permutation
- `symmmd` - Symmetric minimum degree permutation
- `symrcm` - Symmetric reverse Cuthill-McKee permutation

Linear Algebra

- `cholinc` - Incomplete Cholesky factorization
- `condest` - 1-norm condition number estimate
- `eigs` - Eigenvalues and eigenvectors of sparse matrix
- `luinc` - Incomplete LU factorization
- `normest` - Estimate matrix 2-norm
- `sprank` - Structural rank
- `svds` - Singular values and vectors of sparse matrix
### Linear Equations (Iterative Methods)

- **bicg**
  - BiConjugate Gradients method
- **bicgstab**
  - BiConjugate Gradients Stabilized method
- **cgs**
  - Conjugate Gradients Squared method
- **gmres**
  - Generalized Minimum Residual method
- **lsqr**
  - LSQR implementation of Conjugate Gradients on Normal Equations
- **minres**
  - Minimum Residual method
- **pcg**
  - Preconditioned Conjugate Gradients method
- **qmr**
  - Quasi-Minimal Residual method
- **spaugment**
  - Form least squares augmented system
- **symmlq**
  - Symmetric LQ method

### Tree Operations

- **etree**
  - Elimination tree
- **etreeplot**
  - Plot elimination tree
- **gplot**
  - Plot graph, as in “graph theory”
- **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, $\infty$
- **intmax**
  - Largest possible value of specified integer type
- **intmin**
  - Smallest possible value of specified integer type
- **j**
  - Imaginary unit
- **NaN**
  - Not-a-Number
- **pi**
  - Ratio of a circle’s circumference to its diameter, $\pi$
- **realmax**
  - Largest positive floating-point number
- **realmin**
  - Smallest positive floating-point number
Programming and Data Types

Functions to store and operate on data at either the MATLAB command line or in programs and scripts. Functions to write, manage, and execute MATLAB programs.

“Data Types” Numeric, character, structures, cell arrays, and data type conversion

“Arrays” Basic array operations and manipulation

“Operators and Operations” Special characters and arithmetic, bit-wise, relational, logical, set, date and time operations

“Programming in MATLAB” M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling

Data Types

- “Numeric”
- “Characters and Strings”
- “Structures”
- “Cell Arrays”
- “Data Type Conversion”
- “Determine Data Type”
Functions — Categorical List

Numeric

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Array constructor</td>
</tr>
<tr>
<td>cat</td>
<td>Concatenate arrays</td>
</tr>
<tr>
<td>class</td>
<td>Return object’s class name (e.g., numeric)</td>
</tr>
<tr>
<td>find</td>
<td>Find indices and values of nonzero array elements</td>
</tr>
<tr>
<td>intmax</td>
<td>Largest possible value of specified integer type</td>
</tr>
<tr>
<td>intmin</td>
<td>Smallest possible value of specified integer type</td>
</tr>
<tr>
<td>intwarning</td>
<td>Enable or disable integer warnings</td>
</tr>
<tr>
<td>ipermute</td>
<td>Inverse permute dimensions of multidimensional array</td>
</tr>
<tr>
<td>isa</td>
<td>Determine if item is object of given class (e.g., numeric)</td>
</tr>
<tr>
<td>isequal</td>
<td>Determine if arrays are numerically equal</td>
</tr>
<tr>
<td>isequalwithequalnans</td>
<td>Test for equality, treating NaNs as equal</td>
</tr>
<tr>
<td>isnumeric</td>
<td>Determine if item is numeric array</td>
</tr>
<tr>
<td>isscalar</td>
<td>True for scalars (1-by-1 matrices)</td>
</tr>
<tr>
<td>isvector</td>
<td>True for vectors (1-by-N or N-by-1 matrices)</td>
</tr>
<tr>
<td>permute</td>
<td>Rearrange dimensions of multidimensional array</td>
</tr>
<tr>
<td>realmax</td>
<td>Largest positive floating-point number</td>
</tr>
<tr>
<td>realmin</td>
<td>Smallest positive floating-point number</td>
</tr>
<tr>
<td>reshape</td>
<td>Reshape array</td>
</tr>
<tr>
<td>squeeze</td>
<td>Remove singleton dimensions from array</td>
</tr>
<tr>
<td>zeros</td>
<td>Create array of all zeros</td>
</tr>
</tbody>
</table>

Characters and Strings

Description of Strings in MATLAB

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strings</td>
<td>Describes MATLAB string handling</td>
</tr>
</tbody>
</table>

Creating and Manipulating Strings

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blanks</td>
<td>Create string of blanks</td>
</tr>
<tr>
<td>char</td>
<td>Create character array (string)</td>
</tr>
<tr>
<td>cellstr</td>
<td>Create cell array of strings from character array</td>
</tr>
<tr>
<td>datestr</td>
<td>Convert to date string format</td>
</tr>
<tr>
<td>deblank</td>
<td>Strip trailing blanks from the end of string</td>
</tr>
<tr>
<td>lower</td>
<td>Convert string to lower case</td>
</tr>
<tr>
<td>sprintf</td>
<td>Write formatted data to string</td>
</tr>
<tr>
<td>sscanf</td>
<td>Read string under format control</td>
</tr>
<tr>
<td>strcat</td>
<td>String concatenation</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>strjust</td>
<td>Justify character array</td>
</tr>
<tr>
<td>strread</td>
<td>Read formatted data from string</td>
</tr>
<tr>
<td>strrep</td>
<td>String search and replace</td>
</tr>
<tr>
<td>strtrim</td>
<td>Remove leading and trailing whitespace from string</td>
</tr>
<tr>
<td>strvcat</td>
<td>Vertical concatenation of strings</td>
</tr>
<tr>
<td>upper</td>
<td>Convert string to upper case</td>
</tr>
</tbody>
</table>

### Comparing and Searching Strings

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Return object’s class name (e.g., char)</td>
</tr>
<tr>
<td>findstr</td>
<td>Find string within another, longer string</td>
</tr>
<tr>
<td>isa</td>
<td>Determine if item is object of given class (e.g., char)</td>
</tr>
<tr>
<td>iscellstr</td>
<td>Determine if item is cell array of strings</td>
</tr>
<tr>
<td>ischar</td>
<td>Determine if item is character array</td>
</tr>
<tr>
<td>isletter</td>
<td>Detect array elements that are letters of the alphabet</td>
</tr>
<tr>
<td>isscalar</td>
<td>True for scalars (1-by-1 matrices)</td>
</tr>
<tr>
<td>isspace</td>
<td>Detect elements that are ASCII white spaces</td>
</tr>
<tr>
<td>isstrprop</td>
<td>Determine content of each element of string</td>
</tr>
<tr>
<td>isvector</td>
<td>True for vectors (1-by-N or N-by-1 matrices)</td>
</tr>
<tr>
<td>regexp</td>
<td>Match regular expression</td>
</tr>
<tr>
<td>regexpi</td>
<td>Match regular expression, ignoring case</td>
</tr>
<tr>
<td>regexprep</td>
<td>Replace string using regular expression</td>
</tr>
<tr>
<td>strcmp</td>
<td>Compare strings</td>
</tr>
<tr>
<td>strcmpi</td>
<td>Compare strings, ignoring case</td>
</tr>
<tr>
<td>strfind</td>
<td>Find one string within another</td>
</tr>
<tr>
<td>strmatch</td>
<td>Find possible matches for string</td>
</tr>
<tr>
<td>strncmp</td>
<td>Compare first n characters of strings</td>
</tr>
<tr>
<td>strncmpi</td>
<td>Compare first n characters of strings, ignoring case</td>
</tr>
<tr>
<td>strtok</td>
<td>First token in string</td>
</tr>
</tbody>
</table>

### Evaluating String Expressions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell2struct</td>
<td>Cell array to structure array conversion</td>
</tr>
<tr>
<td>class</td>
<td>Return object’s class name (e.g., struct)</td>
</tr>
<tr>
<td>deal</td>
<td>Deal inputs to outputs</td>
</tr>
<tr>
<td>fieldnames</td>
<td>Field names of structure</td>
</tr>
<tr>
<td>isa</td>
<td>Determine if item is object of given class (e.g., struct)</td>
</tr>
<tr>
<td>isequal</td>
<td>Determine if arrays are numerically equal</td>
</tr>
<tr>
<td>isfield</td>
<td>Determine if item is structure array field</td>
</tr>
<tr>
<td>issscalar</td>
<td>True for scalars (1-by-1 matrices)</td>
</tr>
<tr>
<td>isstruct</td>
<td>Determine if item is structure array</td>
</tr>
<tr>
<td>isvector</td>
<td>True for vectors (1-by-N or N-by-1 matrices)</td>
</tr>
<tr>
<td>orderfields</td>
<td>Order fields of a structure array</td>
</tr>
<tr>
<td>rmfield</td>
<td>Remove structure fields</td>
</tr>
<tr>
<td>struct</td>
<td>Create structure array</td>
</tr>
<tr>
<td>struct2cell</td>
<td>Structure to cell array conversion</td>
</tr>
</tbody>
</table>

## Cell Arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>Construct cell array</td>
</tr>
<tr>
<td>cell</td>
<td>Construct cell array</td>
</tr>
<tr>
<td>cellfun</td>
<td>Apply function to each element in cell array</td>
</tr>
<tr>
<td>cellstr</td>
<td>Create cell array of strings from character array</td>
</tr>
<tr>
<td>cell2mat</td>
<td>Convert cell array of matrices into single matrix</td>
</tr>
<tr>
<td>cell2struct</td>
<td>Cell array to structure array conversion</td>
</tr>
<tr>
<td>celldisp</td>
<td>Display cell array contents</td>
</tr>
<tr>
<td>cellplot</td>
<td>Graphically display structure of cell arrays</td>
</tr>
<tr>
<td>class</td>
<td>Return object’s class name (e.g., cell)</td>
</tr>
<tr>
<td>deal</td>
<td>Deal inputs to outputs</td>
</tr>
<tr>
<td>isa</td>
<td>Determine if item is object of given class (e.g., cell)</td>
</tr>
<tr>
<td>iscell</td>
<td>Determine if item is cell array</td>
</tr>
<tr>
<td>iscellstr</td>
<td>Determine if item is cell array of strings</td>
</tr>
<tr>
<td>isequal</td>
<td>Determine if arrays are numerically equal</td>
</tr>
<tr>
<td>issscalar</td>
<td>True for scalars (1-by-1 matrices)</td>
</tr>
<tr>
<td>isvector</td>
<td>True for vectors (1-by-N or N-by-1 matrices)</td>
</tr>
<tr>
<td>mat2cell</td>
<td>Divide matrix up into cell array of matrices</td>
</tr>
<tr>
<td>num2cell</td>
<td>Convert numeric array into cell array</td>
</tr>
<tr>
<td>struct2cell</td>
<td>Structure to cell array conversion</td>
</tr>
</tbody>
</table>
## Data Type Conversion

### Numeric

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double</code></td>
<td>Convert to double-precision</td>
</tr>
<tr>
<td><code>int8</code></td>
<td>Convert to signed 8-bit integer</td>
</tr>
<tr>
<td><code>int16</code></td>
<td>Convert to signed 16-bit integer</td>
</tr>
<tr>
<td><code>int32</code></td>
<td>Convert to signed 32-bit integer</td>
</tr>
<tr>
<td><code>int64</code></td>
<td>Convert to signed 64-bit integer</td>
</tr>
<tr>
<td><code>single</code></td>
<td>Convert to single-precision</td>
</tr>
<tr>
<td><code>uint8</code></td>
<td>Convert to unsigned 8-bit integer</td>
</tr>
<tr>
<td><code>uint16</code></td>
<td>Convert to unsigned 16-bit integer</td>
</tr>
<tr>
<td><code>uint32</code></td>
<td>Convert to unsigned 32-bit integer</td>
</tr>
<tr>
<td><code>uint64</code></td>
<td>Convert to unsigned 64-bit integer</td>
</tr>
</tbody>
</table>

### String to Numeric

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base2dec</code></td>
<td>Convert base N number string to decimal number</td>
</tr>
<tr>
<td><code>bin2dec</code></td>
<td>Convert binary number string to decimal number</td>
</tr>
<tr>
<td><code>hex2dec</code></td>
<td>Convert hexadecimal number string to decimal number</td>
</tr>
<tr>
<td><code>hex2num</code></td>
<td>Convert hexadecimal number string to double number</td>
</tr>
<tr>
<td><code>str2double</code></td>
<td>Convert string to double-precision number</td>
</tr>
<tr>
<td><code>str2num</code></td>
<td>Convert string to number</td>
</tr>
</tbody>
</table>

### Numeric to String

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char</code></td>
<td>Convert to character array (string)</td>
</tr>
<tr>
<td><code>dec2base</code></td>
<td>Convert decimal to base N number in string</td>
</tr>
<tr>
<td><code>dec2bin</code></td>
<td>Convert decimal to binary number in string</td>
</tr>
<tr>
<td><code>dec2hex</code></td>
<td>Convert decimal to hexadecimal number in string</td>
</tr>
<tr>
<td><code>int2str</code></td>
<td>Convert integer to string</td>
</tr>
<tr>
<td><code>mat2str</code></td>
<td>Convert a matrix to string</td>
</tr>
<tr>
<td><code>num2str</code></td>
<td>Convert number to string</td>
</tr>
</tbody>
</table>

### Other Conversions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cell2mat</code></td>
<td>Convert cell array of matrices into single matrix</td>
</tr>
<tr>
<td><code>cell2struct</code></td>
<td>Convert cell array to structure array</td>
</tr>
<tr>
<td><code>datestr</code></td>
<td>Convert serial date number to string</td>
</tr>
<tr>
<td><code>func2str</code></td>
<td>Convert function handle to function name string</td>
</tr>
<tr>
<td><code>logical</code></td>
<td>Convert numeric to logical array</td>
</tr>
<tr>
<td><code>mat2cell</code></td>
<td>Divide matrix up into cell array of matrices</td>
</tr>
<tr>
<td><code>num2cell</code></td>
<td>Convert a numeric array to cell array</td>
</tr>
<tr>
<td><code>str2func</code></td>
<td>Convert function name string to function handle</td>
</tr>
<tr>
<td><code>struct2cell</code></td>
<td>Convert structure to cell array</td>
</tr>
</tbody>
</table>
1

Functions — Categorical List

Determine Data Type
is*
isa
iscell
iscellstr
ischar
isfield
isfloat
isinteger
isjava
islogical
isnumeric
isobject
isreal
isstruct

Detect state
Determine if item is object of given class
Determine if item is cell array
Determine if item is cell array of strings
Determine if item is character array
Determine if item is character array
True for floating-point arrays
True for integer arrays
Determine if item is Java object
Determine if item is logical array
Determine if item is numeric array
Determine if item is MATLAB OOPs object
Determine if all array elements are real numbers
Determine if item is MATLAB structure array

Arrays
• “Array Operations”
• “Basic Array Information”
• “Array Manipulation”
• “Elementary Arrays”

Array Operations
[ ]
,
;
:
end
+
.*
./
.\
.^
.'

1-30

Array constructor
Array row element separator
Array column element separator
Specify range of array elements
Indicate last index of array
Addition or unary plus
Subtraction or unary minus
Array multiplication
Array right division
Array left division
Array power
Array (nonconjugated) transpose


Basic Array Information

disp  Display text or array
display  Overloaded method to display text or array
isempty  Determine if array is empty
isequal  Determine if arrays are numerically equal
isequalwithequalnans  Test for equality, treating NaNs as equal
islogical  Determine if item is logical array
isnumeric  Determine if item is numeric array
isscalar  Determine if item is a scalar
isvector  Determine if item is a vector
length  Length of vector
ndims  Number of array dimensions
numel  Number of elements in matrix or cell array
size  Array dimensions

Array Manipulation

:  Specify range of array elements
blkdiag  Construct block diagonal matrix from input arguments
cat  Concatenate arrays
circshift  Shift array circularly
find  Find indices and values of nonzero elements
fliplr  Flip matrices left-right
flipud  Flip matrices up-down
flipdim  Flip array along specified dimension
horzcat  Horizontal concatenation
ind2sub  Subscripts from linear index
ipermute  Inverse permute dimensions of multidimensional array
permute  Rearrange dimensions of multidimensional 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
sub2ind  Single index from subscripts
vertcat  Horizontal concatenation
**Elementary Arrays**

: Regularly spaced vector  
`blkdiag` Construct block diagonal matrix from input arguments  
`eye` Identity matrix  
`linspace` Generate linearly spaced vectors  
`logspace` Generate logarithmically spaced vectors  
`meshgrid` Generate X and Y matrices for three-dimensional plots  
`ndgrid` Generate arrays for multidimensional functions and interpolation  
`ones` Create array of all ones  
`rand` Uniformly distributed random numbers and arrays  
`randn` Normally distributed random numbers and arrays  
`zeros` Create array of all zeros

**Operators and Operations**

- “Special Characters”
- “Arithmetic Operations”
- “Bit-wise Operations”
- “Relational Operations”
- “Logical Operations”
- “Set Operations”
- “Date and Time Operations”

**Special Characters**

: Specify range of array elements  
( ) Pass function arguments, or prioritize operations  
[ ] Construct array  
{ } Construct cell array  
. Decimal point, or structure field separator  
... Continue statement to next line  
, Array row element separator  
; Array column element separator  
% Insert comment line into code  
! Command to operating system  
= Assignment
### Arithmetic Operations

+  
   **Plus**

-  
   **Minus**

.  
   **Decimal point**

=  
   **Assignment**

*  
   **Matrix multiplication**

/  
   **Matrix right division**

\  
   **Matrix left division**

^  
   **Matrix power**

'  
   **Matrix transpose**

.*  
   **Array multiplication (element-wise)**

./  
   **Array right division (element-wise)**

./\  
   **Array left division (element-wise)**

.^  
   **Array power (element-wise)**

.'  
   **Array transpose**

### Bit-wise Operations

bitand  
   **Bit-wise AND**

bitcmp  
   **Bit-wise complement**

bitor  
   **Bit-wise OR**

bitmax  
   **Maximum floating-point integer**

bitset  
   **Set bit at specified position**

bitshift  
   **Bit-wise shift**

bitget  
   **Get bit at specified position**

bitxor  
   **Bit-wise XOR**

### Relational Operations

<  
   **Less than**

<=  
   **Less than or equal to**

>  
   **Greater than**

>=  
   **Greater than or equal to**

==  
   **Equal to**

~=  
   **Not equal to**
**Logical Operations**

- `&&`: Logical AND
- `||`: Logical OR
- `&`: Logical AND for arrays
- `|`: Logical OR for arrays
- `~`: Logical NOT
- `all`: Test to determine if all elements are nonzero
- `any`: Test for any nonzero elements
- `false`: False array
- `find`: Find indices and values of nonzero elements
- `is*`: Detect state
- `isa`: Determine if item is object of given class
- `iskeyword`: Determine if string is MATLAB keyword
- `isvarname`: Determine if string is valid variable name
- `logical`: Convert numeric values to logical
- `true`: True array
- `xor`: Logical EXCLUSIVE OR

**Set Operations**

- `intersect`: Set intersection of two vectors
- `ismember`: Detect members of set
- `setdiff`: Return set difference of two vectors
- `issorted`: Determine if set elements are in sorted order
- `setxor`: Set exclusive or of two vectors
- `union`: Set union of two vectors
- `unique`: Unique elements of vector

**Date and Time Operations**

- `addtodate`: Modify particular field of date number
- `calendar`: Calendar for specified month
- `clock`: Current time as date vector
- `cputime`: Elapsed CPU time
- `date`: Current date string
- `datenum`: Serial date number
- `datestr`: Convert serial date number to string
- `datevec`: Date components
- `eomday`: End of month
- `etime`: Elapsed time
- `now`: Current date and time
- `tic, toc`: Stopwatch timer
- `weekday`: Day of the week
Programming in MATLAB

- “M-File Functions and Scripts”
- “Evaluation of Expressions and Functions”
- “Timer Functions”
- “Variables and Functions in Memory”
- “Control Flow”
- “Function Handles”
- “Object-Oriented Programming”
- “Error Handling”
- “MEX Programming”

M-File Functions and Scripts

( ) Pass function arguments
% Insert comment line into code
... Continue statement to next line
depfun List dependent functions of M-file or P-file
depdir List dependent directories of M-file or P-file
echo Echo M-files during execution
function Function M-files
input Request user input
inputname Input argument name
mfilename Name of currently running M-file
nameengthmax Return maximum identifier length
nargin Number of function input arguments
nargout Number of function output arguments
nargchk Check number of input arguments
nargoutchk Validate number of output arguments
pcode Create preparsed pseudocode file (P-file)
script Describes script M-file
varargin Accept variable number of arguments
varargout Return variable number of arguments
### Evaluation of Expressions and Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>builtin</code></td>
<td>Execute built-in function from overloaded method</td>
</tr>
<tr>
<td><code>cellfun</code></td>
<td>Apply function to each element in cell array</td>
</tr>
<tr>
<td><code>echo</code></td>
<td>Echo M-files during execution</td>
</tr>
<tr>
<td><code>eval</code></td>
<td>Interpret strings containing MATLAB expressions</td>
</tr>
<tr>
<td><code>evalc</code></td>
<td>Evaluate MATLAB expression with capture</td>
</tr>
<tr>
<td><code>evalin</code></td>
<td>Evaluate expression in workspace</td>
</tr>
<tr>
<td><code>feval</code></td>
<td>Evaluate function</td>
</tr>
<tr>
<td><code>iskeyword</code></td>
<td>Determine if item is MATLAB keyword</td>
</tr>
<tr>
<td><code>isvarname</code></td>
<td>Determine if item is valid variable name</td>
</tr>
<tr>
<td><code>pause</code></td>
<td>Halt execution temporarily</td>
</tr>
<tr>
<td><code>run</code></td>
<td>Run script that is not on current path</td>
</tr>
<tr>
<td><code>script</code></td>
<td>Describes script M-file</td>
</tr>
<tr>
<td><code>symvar</code></td>
<td>Determine symbolic variables in expression</td>
</tr>
<tr>
<td><code>tic, toc</code></td>
<td>Stopwatch timer</td>
</tr>
</tbody>
</table>

### Timer Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>delete</code></td>
<td>Delete timer object from memory</td>
</tr>
<tr>
<td><code>disp</code></td>
<td>Display information about timer object</td>
</tr>
<tr>
<td><code>get</code></td>
<td>Retrieve information about timer object properties</td>
</tr>
<tr>
<td><code>isvalid</code></td>
<td>Determine if timer object is valid</td>
</tr>
<tr>
<td><code>set</code></td>
<td>Display or set timer object properties</td>
</tr>
<tr>
<td><code>start</code></td>
<td>Start a timer</td>
</tr>
<tr>
<td><code>startat</code></td>
<td>Start a timer at a specific timer</td>
</tr>
<tr>
<td><code>stop</code></td>
<td>Stop a timer</td>
</tr>
<tr>
<td><code>timer</code></td>
<td>Create a timer object</td>
</tr>
<tr>
<td><code>timerfind</code></td>
<td>Return an array of all visible timer objects in memory</td>
</tr>
<tr>
<td><code>timerfindall</code></td>
<td>Return an array of all timer objects in memory</td>
</tr>
<tr>
<td><code>wait</code></td>
<td>Block command line until timer completes</td>
</tr>
</tbody>
</table>

### Variables and Functions in Memory

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assignin</code></td>
<td>Assign value to workspace variable</td>
</tr>
<tr>
<td><code>genvarname</code></td>
<td>Construct valid variable name from string</td>
</tr>
<tr>
<td><code>global</code></td>
<td>Define global variables</td>
</tr>
<tr>
<td><code>inmem</code></td>
<td>Return names of functions in memory</td>
</tr>
<tr>
<td><code>isglobal</code></td>
<td>Determine if item is global variable</td>
</tr>
<tr>
<td><code>mislocked</code></td>
<td>True if M-file cannot be cleared</td>
</tr>
<tr>
<td><code>mlock</code></td>
<td>Prevent clearing M-file from memory</td>
</tr>
<tr>
<td><code>munlock</code></td>
<td>Allow clearing M-file from memory</td>
</tr>
<tr>
<td><code>namelengthmax</code></td>
<td>Return maximum identifier length</td>
</tr>
<tr>
<td><code>pack</code></td>
<td>Consolidate workspace memory</td>
</tr>
<tr>
<td><code>persistent</code></td>
<td>Define persistent variable</td>
</tr>
<tr>
<td><code>rehash</code></td>
<td>Refresh function and file system caches</td>
</tr>
</tbody>
</table>
Control Flow

- **break**: Terminate execution of `for` loop or `while` loop
- **case**: Case switch
- **catch**: Begin catch block
- **continue**: Pass control to next iteration of `for` or `while` loop
- **else**: Conditionally execute statements
- **elseif**: Conditionally execute statements
- **end**: Terminate conditional statements, or indicate last index
- **error**: Display error messages
- **for**: Repeat statements specific number of times
- **if**: Conditionally execute statements
- **otherwise**: Default part of `switch` statement
- **return**: Return to invoking function
- **switch**: Switch among several cases based on expression
- **try**: Begin `try` block
- **while**: Repeat statements indefinite number of times

Function Handles

- **class**: Return object’s class name (e.g. `function_handle`)
- **feval**: Evaluate function
- **function_handle**: Describes function handle data type
- **functions**: Return information about function handle
- **func2str**: Constructs function name string from function handle
- **isa**: Determine if item is object of given class (e.g. `function_handle`)
- **isequal**: Determine if function handles are equal
- **str2func**: Constructs function handle from function name string

Object-Oriented Programming

MATLAB Classes and Objects

- **class**: Create object or return class of object
- **fieldnames**: List public fields belonging to object,
- **inferiorto**: Establish inferior class relationship
- **isa**: Determine if item is object of given class
- **isobject**: Determine if item is MATLAB OOPs object
- **loadobj**: User-defined extension of `load` function for user objects
- **methods**: Display information on class methods
- **methodsview**: Display information on class methods in separate window
- **saveobj**: User-defined extension of `save` function for user objects
- **subsasgn**: Overloaded method for `A(I)=B`, `A{I}=B`, and `A.field=B`
subsindex Overloaded method for X(A)
subsref Overloaded method for A(I), A{I} and A.field
substruct Create structure argument for subsasgn or subsref
superiorto Establish superior class relationship

Java Classes and Objects

cell Convert Java array object to cell array
class Return class name of Java object
clear Clear Java import list or Java class definitions
depfun List Java classes used by M-file
eexist Determine if item is Java class
fieldnames List public fields belonging to object
im2java Convert image to instance of Java image object
import Add package or class to current Java import list
inmem List names of Java classes loaded into memory
isa Determine if item is object of given class
isjava Determine if item is Java object
javaaddpath Add entries to dynamic Java class path
javaArray Construct Java array
javachk Generate error message based on Java feature support
javaclasspath Set and get dynamic Java class path
javamethod Invoke Java method
javaobject Construct Java object
javarmpath Remove entries from dynamic Java class path
methods Display information on class methods
methodsviw Display information on class methods in separate window
usejava Determine if a Java feature is supported in MATLAB
which Display package and class name for method

Error Handling

catch Begin catch block of try/catch statement
eerror Display error message
eferror Query MATLAB about errors in file input or output
intwarning Enable or disable integer warnings
lasterr Return last error message generated by MATLAB
lasterror Last error message and related information
lastwarn Return last warning message issued by MATLAB
rethrow Reissue error
try Begin try block of try/catch statement
warning Display warning message
**MEX Programming**

- `dbmex`: Enable MEX-file debugging
- `inmem`: Return names of currently loaded MEX-files
- `mex`: Compile MEX-function from C or Fortran source code
- `mexext`: Return MEX-filename extension
File I/O

Functions to read and write data to files of different format types.

“Filename Construction” Get path, directory, filename information; construct filenames
“Opening, Loading, Saving Files” Open files; transfer data between files and MATLAB workspace
“Low-Level File I/O” Low-level operations that use a file identifier (e.g., fopen, fseek, fread)
“Text Files” Delimited or formatted I/O to text files
“XML Documents” Documents written in Extensible Markup Language
“Spreadsheets” Excel and Lotus 123 files
“Scientific Data” CDF, FITS, HDF formats
“Audio and Audio/Video” General audio functions; SparcStation, WAVE, AVI files
“Images” Graphics files
“Internet Exchange” URL, zip, and e-mail

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

Filename Construction

fileparts Return parts of filename
filesep Return directory separator for this platform
fullfile Build full filename from parts
tempdir Return name of system's temporary directory
tempname Return unique string for use as temporary filename
Opening, Loading, Saving Files

importdata    Load data from various types of files
load          Load all or specific data from MAT or ASCII file
open          Open files of various types using appropriate editor or program
save          Save all or specific data to MAT or ASCII file
uiimport      Open Import Wizard, the graphical user interface to import data
winopen       Open file in appropriate application (Windows only)

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
fgets          Return next line of file as string with line terminator(s)
fprintf        Write formatted data to file
fgetl          Return next line of file as string without line terminator(s)
fwrite         Write binary data to file
fopen          Open file or obtain information about open files
fread          Read binary data from file
frewind        Rewind open file
fscanf         Read formatted data from file
fseek          Set file position indicator
fseek          Get file position indicator
fwrite         Write binary data to file

Text Files

csvread        Read numeric data from text file, using comma delimiter
csvwrite       Write numeric data to text file, using comma delimiter
dlmread        Read numeric data from text file, specifying your own delimiter
dlmwrite       Write numeric data to text file, specifying your own delimiter
textread       Read data from text file, write to multiple outputs
textscan       Read data from text file, convert and write to cell array

XML Documents

xmlread        Parse XML document
xmlwrite       Serialize XML Document Object Model node
xslt           Transform XML document using XSLT engine
Spreadsheets

Microsoft Excel Functions
- xlsfinfo: Determine if file contains Microsoft Excel (.xls) spreadsheet
- xlsread: Read Microsoft Excel spreadsheet file (.xls)
- x1swrite: Write Microsoft Excel spreadsheet file (.xls)

Lotus123 Functions
- wk1read: Read Lotus123 WK1 spreadsheet file into matrix
- wk1write: Write matrix to Lotus123 WK1 spreadsheet file

Scientific Data

Common Data Format (CDF)
- cdfepoch: Convert MATLAB date number or date string into CDF epoch
- cdfinfo: Return information about CDF file
- cdfread: Read CDF file
- cdfwrite: Write CDF file

Flexible Image Transport System
- fitsinfo: Return information about FITS file
- fitsread: Read FITS file

Hierarchical Data Format (HDF)
- hdf: Interface to HDF4 files
- hdfinfo: Return information about HDF4 or HDF-EOS file
- hdfread: Read HDF4 file
- hdftool: Start HDF4 Import Tool
- hdf5: Describes HDF5 data type objects
- hdf5info: Return information about HDF5 file
- hdf5read: Read HDF5 file
- hdf5write: Write data to file in HDF5 format

Band-Interleaved Data
- multibandread: Read band-interleaved data from file
- multibandwrite: Write band-interleaved data to file
Audio and Audio/Video

General
audioplayer Create audio player object
audiorecorder Perform real-time audio capture
beep Produce beep sound
lin2mu Convert linear audio signal to mu-law
mmfileinfo Information about a 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
auread Read NeXT/SUN (.au) sound file
auwrite Write NeXT/SUN (.au) sound file

Microsoft WAVE Sound Functions
wavplay Play sound on PC-based audio output device
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 AVI file
avifile Create new AVI file
aviinfo Return information about AVI file
aviread Read AVI file
close Close AVI file
movie2avi Create AVI movie from MATLAB movie

Images
im2java Convert image to instance of Java image object
imfinfo Return information about graphics file
imread Read image from graphics file
imwrite Write image to graphics file
Internet Exchange

- **ftp**: Connect to FTP server, creating an FTP object
- **sendmail**: Send e-mail message (attachments optional) to list of addresses
- **unzip**: Extract contents of zip file
- **urlread**: Read contents at URL
- **urlwrite**: Save contents of URL to file
- **zip**: Create compressed version of files in zip format
Graphics

2-D graphs, specialized plots (e.g., pie charts, histograms, and contour plots), function plotters, and Handle Graphics functions.

Basic Plots and Graphs
- box
  - Axis box for 2-D and 3-D plots
- errorbar
  - Plot graph with error bars
- hold
  - Hold current graph
- LineSpec
  - Line specification syntax
- loglog
  - Plot using log-log scales
- polar
  - Polar coordinate plot
- plot
  - Plot vectors or matrices.
- plot3
  - Plot lines and points in 3-D space
- plotyy
  - Plot graphs with Y tick labels on the left and right
- semilogx
  - Semi-log scale plot
- semilogy
  - Semi-log scale plot
- subplot
  - Create axes in tiled positions

Plotting Tools
- figurepalette
  - Display figure palette on figure
- pan
  - Turn panning on or off
- plotbrowser
  - Display plot browser on figure
- plottools
  - Start plotting tools
- propertyeditor
  - Display property editor on figure
- zoom
  - Turn zooming on or off

Annotating Plots
- Titles, axes labels, legends, mathematical symbols

Specialized Plotting
- Bar graphs, histograms, pie charts, contour plots, function plotters

Bit-Mapped Images
- Display image object, read and write graphics file, convert to movie frames

Printing
- Printing and exporting figures to standard formats

Handle Graphics
- Creating graphics objects, setting properties, finding handles
Annotating Plots

- `annotation`: Create annotation objects
- `clabel`: Add contour labels to contour plot
- `datetick`: Date formatted tick labels
- `gtext`: Place text on 2-D graph using mouse
- `legend`: Graph legend for lines and patches
- `texlabel`: Produce the TeX format from character string
- `title`: Titles for 2-D and 3-D plots
- `xlabel`: X-axis labels for 2-D and 3-D plots
- `ylabel`: Y-axis labels for 2-D and 3-D plots
- `zlabel`: Z-axis labels for 3-D plots

Annotation Object Properties

- `arrow`: Properties for annotation arrows
- `doublearrow`: Properties for double-headed annotation arrows
- `ellipse`: Properties for annotation ellipses
- `line`: Properties for annotation lines
- `rectangle`: Properties for annotation rectangles
- `textarrow`: Properties for annotation textbox

Specialized Plotting

- “Area, Bar, and Pie Plots”
- “Contour Plots”
- “Direction and Velocity Plots”
- “Discrete Data Plots”
- “Function Plots”
- “Histograms”
- “Polygons and Surfaces”
- “Scatter/Bubble Plots”
- “Animation”
Area, Bar, and Pie Plots

- **area**: Area plot
- **bar**: Vertical bar chart
- **barch**: Horizontal bar chart
- **bar3**: Vertical 3-D bar chart
- **bar3h**: Horizontal 3-D bar chart
- **pareto**: Pareto chart
- **pie**: Pie plot
- **pie3**: 3-D pie plot

Contour Plots

- **contour**: Contour (level curves) plot
- **contour3**: 3-D contour plot
- **contourc**: Contour computation
- **contourf**: Filled contour plot
- **ezcontour**: Easy to use contour plotter
- **ezcontourf**: Easy to use filled contour plotter

Direction and Velocity Plots

- **comet**: Comet plot
- **comet3**: 3-D comet plot
- **compass**: Compass plot
- **feather**: Feather plot
- **quiver**: Quiver (or velocity) plot
- **quiver3**: 3-D quiver (or velocity) plot

Discrete Data Plots

- **stem**: Plot discrete sequence data
- **stem3**: Plot discrete surface data
- **stairs**: Stairstep graph

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
- **ezsurf3**: Easy to use combination surface/contour plotter
- **fplot**: Plot a function
### Histograms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hist</td>
<td>Plot histograms</td>
</tr>
<tr>
<td>histc</td>
<td>Histogram count</td>
</tr>
<tr>
<td>rose</td>
<td>Plot rose or angle histogram</td>
</tr>
</tbody>
</table>

### Polygons and Surfaces

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convhull</td>
<td>Convex hull</td>
</tr>
<tr>
<td>cylinder</td>
<td>Generate cylinder</td>
</tr>
<tr>
<td>delaunay</td>
<td>Delaunay triangulation</td>
</tr>
<tr>
<td>dsearch</td>
<td>Search Delaunay triangulation for nearest point</td>
</tr>
<tr>
<td>ellipsoid</td>
<td>Generate ellipsoid</td>
</tr>
<tr>
<td>fill</td>
<td>Draw filled 2-D polygons</td>
</tr>
<tr>
<td>fill3</td>
<td>Draw filled 3-D polygons in 3-space</td>
</tr>
<tr>
<td>inpolyon</td>
<td>True for points inside a polygonal region</td>
</tr>
<tr>
<td>pcolor</td>
<td>Pseudocolor (checkerboard) plot</td>
</tr>
<tr>
<td>polyarea</td>
<td>Area of polygon</td>
</tr>
<tr>
<td>ribbon</td>
<td>Ribbon plot</td>
</tr>
<tr>
<td>slice</td>
<td>Volumetric slice plot</td>
</tr>
<tr>
<td>sphere</td>
<td>Generate sphere</td>
</tr>
<tr>
<td>tsearch</td>
<td>Search for enclosing Delaunay triangle</td>
</tr>
<tr>
<td>voronoi</td>
<td>Voronoi diagram</td>
</tr>
<tr>
<td>waterfall</td>
<td>Waterfall plot</td>
</tr>
</tbody>
</table>

### Scatter/Bubble Plots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plotmatrix</td>
<td>Scatter plot matrix</td>
</tr>
<tr>
<td>scatter</td>
<td>Scatter plot</td>
</tr>
<tr>
<td>scatter3</td>
<td>3-D scatter plot</td>
</tr>
</tbody>
</table>

### Animation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame2im</td>
<td>Convert movie frame to indexed image</td>
</tr>
<tr>
<td>getframe</td>
<td>Capture movie frame</td>
</tr>
<tr>
<td>im2frame</td>
<td>Convert image to movie frame</td>
</tr>
<tr>
<td>movie</td>
<td>Play recorded movie frames</td>
</tr>
<tr>
<td>noanimate</td>
<td>Change EraseMode of all objects to normal</td>
</tr>
</tbody>
</table>
Bit-Mapped Images

- frame2im: Convert movie frame to indexed image
- image: Display image object
- imagesc: Scale data and display image object
- imfinfo: Information about graphics file
- imformats: Manage file format registry
- im2frame: Convert image to movie frame
- im2java: Convert image to instance of Java image object
- imread: Read image from graphics file
- imwrite: Write image to graphics file
- ind2rgb: Convert indexed image to RGB image

Printing

- frameedit: Edit print frame for Simulink and Stateflow diagram
- orient: Hardcopy paper orientation
- pagesetupdlg: Page setup dialog box
- print: Print graph or save graph to file
- printdlg: Print dialog box
- printopt: Configure local printer defaults
- printpreview: Preview figure to be printed
- saveas: Save figure to graphic file

Handle Graphics

- Finding and Identifying Graphics Objects
- Object Creation Functions
- Figure Windows
- Axes Operations
Finding and Identifying Graphics Objects

- **allchild**: Find all children of specified objects
- **ancestor**: Find ancestor of graphics object
- **copyobj**: Make copy of graphics object and its children
- **delete**: Delete files or graphics objects
- **findall**: Find all graphics objects (including hidden handles)
- **figflag**: Test if figure is on screen
- **findfigs**: Display off-screen visible figure windows
- **findobj**: Find objects with specified property values
- **gca**: Get current Axes handle
- **gcbo**: Return object whose callback is currently executing
- **gcbf**: Return handle of figure containing callback object
- **gco**: Return handle of current object
- **get**: Get object properties
- **ishandle**: True if value is valid object handle
- **set**: Set object properties

Object Creation Functions

- **axes**: Create axes object
- **figure**: Create figure (graph) windows
- **hggroup**: Create a group object
- **hgtransform**: Create a group to transform
- **image**: Create image (2-D matrix)
- **light**: Create light object (illuminates Patch and Surface)
- **line**: Create line object (3-D polylines)
- **patch**: Create patch object (polygons)
- **rectangle**: Create rectangle object (2-D rectangle)
- **rootobject**: List of root properties
- **surface**: Create surface (quadrilaterals)
- **text**: Create text object (character strings)
- **uicontextmenu**: Create context menu (popup associated with object)

Plot Objects

- **areaseries**: Property list
- **barseries**: Property list
- **contourgroup**: Property list
- **errorbarsseries**: Property list
- **lineseries**: Property list
- **quivergroup**: Property list
- **scattergroup**: Property list
- **stairseries**: Property list
- **stemseries**: Property list
- **surfaceplot**: Property list
Figure Windows
clc  Clear figure window
clf  Clear figure
close Close specified window
closereq Default close request function
drawnow Complete any pending drawing
figflag Test if figure is on screen
gcf  Get current figure handle
hgload Load graphics object hierarchy from a FIG-file
hgsave Save graphics object hierarchy to a FIG-file
newplot Graphics M-file preamble for NextPlot property
opengl Change automatic selection mode of OpenGL rendering
refresh Refresh figure
saveas Save figure or model to desired output format

Axes Operations
axis Plot axis scaling and appearance
box Display axes border
cla Clear Axes
gca Get current Axes handle
grid Grid lines for 2-D and 3-D plots
ishold Get the current hold state
makehgtform Create a transform matrix

Operating on Object Properties
get Get object properties
linkaxes Synchronize limits of specified axes
linkprop Maintain same value for corresponding properties
set Set object properties
3-D Visualization

Create and manipulate graphics that display 2-D matrix and 3-D volume data, controlling the view, lighting and transparency.

Surface and Mesh Plots
- Plot matrices, visualize functions of two variables, specify colormap

View Control
- Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits

Lighting
- Add and control scene lighting

Transparency
- Specify and control object transparency

Volume Visualization
- Visualize gridded volume data

Surface and Mesh Plots

- Creating Surfaces and Meshes
- Domain Generation
- Color Operations
- Colormaps

Creating Surfaces and Meshes
- hidden: Mesh hidden line removal mode
- meshc: Combination mesh/contourplot
- mesh: 3-D mesh with reference plane
- peaks: A sample function of two variables
- surf: 3-D shaded surface graph
- surface: Create surface low-level objects
- surfc: Combination surf/contourplot
- surfl: 3-D shaded surface with lighting
- tetramesh: Tetrahedron mesh plot
- trimesh: Triangular mesh plot
- triplot: 2-D triangular plot
- trisurf: Triangular surface plot

Domain Generation
- griddata: Data gridding and surface fitting
- meshgrid: Generation of X and Y arrays for 3-D plots
Color Operations

- **brighten**: Brighten or darken colormap
- **caxis**: Pseudocolor axis scaling
- **colormapeditor**: Start colormap editor
- **colorbar**: Display color bar (color scale)
- **colordef**: Set up color defaults
- **colormap**: Set the color look-up table (list of colormaps)
- **ColorSpec**: Ways to specify color
- **graymon**: Graphics figure defaults set for grayscale monitor
- **hsv2rgb**: Hue-saturation-value to red-green-blue conversion
- **rgb2hsv**: RGB to HSV conversion
- **rgbplot**: Plot colormap
- **shading**: Color shading mode
- **spinmap**: Spin the colormap
- **surfnormal**: 3-D surface normals
- **whitebg**: Change axes background color for plots

Colormaps

- **autumn**: Shades of red and yellow colormap
- **bone**: Gray-scale with a tinge of blue colormap
- **contrast**: Gray colormap to enhance image contrast
- **cool**: Shades of cyan and magenta colormap
- **copper**: Linear copper-tone colormap
- **flag**: Alternating red, white, blue, and black colormap
- **gray**: Linear gray-scale colormap
- **hot**: Black-red-yellow-white colormap
- **hsv**: Hue-saturation-value (HSV) colormap
- **jet**: Variant of HSV
- **lines**: Line color colormap
- **prism**: Colormap of prism colors
- **spring**: Shades of magenta and yellow colormap
- **summer**: Shades of green and yellow colormap
- **winter**: Shades of blue and green colormap

View Control

- Controlling the Camera Viewpoint
- Setting the Aspect Ratio and Axis Limits
- Object Manipulation
- Selecting Region of Interest
Controlling the Camera Viewpoint

camdolly  Move camera position and target
camlookat View specific objects
camorbit  Orbit about camera target
campan   Rotate camera target about camera position
campos   Set or get camera position
camproj  Set or get projection type
camroll  Rotate camera about viewing axis
camtarget Set or get camera target
cameratoolbar Control camera toolbar programmatically
camup    Set or get camera up-vector
camva    Set or get camera view angle
camzoom  Zoom camera in or out
view     3-D graph viewpoint specification.
viewmtx Generate view transformation matrices
makehgtform Create a transform matrix

Setting the Aspect Ratio and Axis Limits

daspect  Set or get data aspect ratio
pbaspect Set or get plot box aspect ratio
xlim     Set or get the current x-axis limits
ylim     Set or get the current y-axis limits
zlim     Set or get the current z-axis limits

Object Manipulation

pan       Turns panning on or off
reset    Reset axis or figure
rotate   Rotate objects about specified origin and direction
rotate3d  Interactively rotate the view of a 3-D plot
selectmoveresize Interactively select, move, or resize objects
zoom     Zoom in and out on a 2-D plot

Selecting Region of Interest

dragrect   Drag XOR rectangles with mouse
rbbox     Rubberband box
**Lighting**
camlight  Cerate or position Light
light      Light object creation function
lightangle Position light in spherical coordinates
lighting   Lighting mode
material   Material reflectance mode

**Transparency**
alpha      Set or query transparency properties for objects in current axes
alphamap   Specify the figure alphamap
alim       Set or query the axes alpha limits

**Volume Visualization**
coneplot   Plot velocity vectors as cones in 3-D vector field
contourslice Draw contours in volume slice plane
curl       Compute curl and angular velocity of vector field
divergence Compute divergence of vector field
flow       Generate scalar volume data
interpstreamspeed Interpolate streamline vertices from vector-field magnitudes
isocaps    Compute isosurface end-cap geometry
iscolors   Compute colors of isosurface vertices
isonormals Compute normals of isosurface vertices
isosurface Extract isosurface data from volume data
reducepatch Reduce number of patch faces
reducevolume Reduce number of elements in volume data set
shrinkfaces Reduce size of patch faces
slice      Draw slice planes in volume
smooth3    Smooth 3-D data
stream2    Compute 2-D stream line data
stream3    Compute 3-D stream line data
streamline Draw stream lines from 2- or 3-D vector data
streamparticles Draws stream particles from vector volume data
streamribbon Draws stream ribbons from vector volume data
streamslice Draws well-spaced stream lines from vector volume data
streamtube Draws stream tubes from vector volume data
surf2patch Convert surface data to patch data
subvolume   Extract subset of volume data set
volumebounds Return coordinate and color limits for volume (scalar and vector)
Creating Graphical User Interfaces

Predefined dialog boxes and functions to control GUI programs.

Predefined Dialog Boxes
- dialog: Create dialog box
- errordlg: Create error dialog box
- helpdlg: Display help dialog box
- inputdlg: Create input dialog box
- listdlg: Create list selection dialog box
- msgbox: Create message dialog box
- pagesetupdlg: Page setup dialog box
- printdlg: Display print dialog box
- questdlg: Create question dialog box
- uigetdir: Display dialog box to retrieve name of directory
- uigetfile: Display dialog box to retrieve name of file for reading
- uiputfile: Display dialog box to retrieve name of file for writing
- uisetcolor: Set ColorSpec using dialog box
- uisetfont: Set font using dialog box
- waitbar: Display wait bar
- warndlg: Create warning dialog box

Deploying User Interfaces
- Launching GUIs, creating the handles structure

Developing User Interfaces
- Starting GUIDE, managing application data, getting user input

User Interface Objects
- Creating GUI components

Finding Objects from Callbacks
- Finding object handles from within callbacks functions

GUI Utility Functions
- Moving objects, text wrapping

Controlling Program Execution
- Wait and resume based on user input
Deploying User Interfaces

guihandles Create a structure of handles
movegui Move GUI figure onscreen
openfig Open or raise GUI figure

Developing User Interfaces

guide Open GUI Layout Editor
inspect Display Property Inspector

Working with Application Data

getappdata Get value of application data
isappdata True if application data exists
rmappdata Remove application data
setappdata Specify application data

Interactive User Input

ginput Graphical input from a mouse or cursor
waitfor Wait for conditions before resuming execution
waitforbuttonpress Wait for key/buttonpress over figure

User Interface Objects

menu Generate menu of choices for user input
uibuttongroup Create component to exclusively manage radio buttons and toggle buttons
uicontextmenu Create context menu
uicontrol Create user interface control
uimenu Create user interface menu
uipanel Create panel container object
uipushtool Create toolbar push button
uitoggletool Create toolbar toggle button
uitoolbar Create toolbar

Finding Objects from Callbacks

findall Find all graphics objects
findfigs Display off-screen visible figure windows
findobj Find specific graphics object
gcbf Return handle of figure containing callback object
gcbo Return handle of object whose callback is executing
Functions — Alphabetical List
pack

2pack

**Purpose**
Consolidate workspace memory

**Syntax**
- `pack`
- `pack filename`
- `pack('filename')`

**Description**
`pack` frees up needed space by reorganizing information so it only uses the minimum memory required. You must run `pack` from a directory for which you have write permission. Running `pack` clears all variables not in the base workspace, so persistent variables, for example, will be cleared.

`pack filename` accepts an optional `filename` for the temporary file used to hold the variables. Otherwise, it uses the file named `pack.tmp`. You must run `pack` from a directory for which you have write permission.

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

**Remarks**
The `pack` function does not affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory.

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the Out of memory message from MATLAB, the `pack` function may find you some free memory without forcing you to delete variables.

The `pack` function frees space by:

- Saving all variables in the base workspace to disk in a temporary file called `pack.tmp`
- Clearing all variables and functions from memory
- Reloading the base workspace variables back from `pack.tmp`
- Deleting the temporary file `pack.tmp`
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
pagesetupdlg

Purpose
Page position dialog box

Syntax

dlg = pagesetupdlg(fig)

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.

Unlike pagedlg, pagesetupdlg currently only 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
printpreview, printopt
Purpose
Pan the view of a graph interactively

Syntax
pan on
pan xon
pan yon
pan off
pan
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.

See Also
zoom, linkaxes

“Object Manipulation” for related functions
**Purpose**
Pareto chart

**Syntax**
```
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.

- `pareto(Y)` labels each bar with its element index in 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.
- `H = pareto(...)` returns a combination of patch and line object handles.

**See Also**
hist, bar
Purpose
Partial pathname

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 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`, and `fopen`

Examples
The following example uses a partial pathname:

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

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

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

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

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

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

```matlab
what graph2d/figobj
```

See Also
`fileparts`, `matlabroot`, `path`
pascal

Purpose
Pascal matrix

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

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

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

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

Examples
pascal(4) returns

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

A = pascal(3,2) produces

A =

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-2</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

See Also
chol
Purpose

Create patch graphics object

Syntax

patch(X,Y,C)
patch(X,Y,Z,C)
patch(FV)
patch(...'PropertyName',PropertyValue...)
patch('PropertyName',PropertyValue...) PN/PV pairs only
handle = patch(...)

Description

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

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

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

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

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

patch('PropertyName',PropertyValue,...) specifies all properties using property name/property value pairs. This form enables you to omit the color specification because MATLAB uses the default face color and edge color unless you explicitly assign a value to the FaceColor and EdgeColor properties. This form also allows you to specify the patch using the Faces and Vertices properties instead of x-, y-, and z-coordinates. See the “Examples” 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.
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>[X,Y,Z]Data Dimensions</th>
<th>CData Required for 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>
</tbody>
</table>
Interpretation of the FaceVertexCData Property

| X,Y,Z| CData Required for | Results Obtained                                                                 |
| Dimensions | Indexed | True Color |                               |
| m-by-n | 1-by-n | 1-by-n-by-3 | Use one color for each patch face. Edges can be only a single color. |
| m-by-n | m-by-n | m-by-n-3 | Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated. |

Interpretation of the FaceVertexCData Property

| Vertices | Faces | FaceVertexCData | Results Obtained                                                                 |
| Dimensions | Dimensions | Required for | Indexed | True Color |                               |
| m-by-n | k-by-3 | scalar | 1-by-3 | Use the single color specified for all patch faces. Edges can be only a single color. |
| m-by-n | k-by-3 | k-by-1 | k-by-3 | Use one color for each patch face. Edges can be only a single color. |
| m-by-n | k-by-3 | m-by-1 | m-by-3 | Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated. |

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.

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

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

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

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

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

tcolor = [1 1 1; .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.

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

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

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the `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**

```
Axes
  Group
    Object
    Patch
    Patch
```

**Setting Default Properties**

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

```matlab
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.

**Property List**

The following table lists all patch properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Defining the Object</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Faces | Connection matrix for Vertices | Values: m-by-n matrix  
Default: [1,2,3] |
| Vertices | Matrix of x-, y-, and z-coordinates of the vertices (used with Faces) | Value: matrix  
Default: [0,1;1,1;0,0] |
| XData | The x-coordinates of the vertices of the patch | Value: vector or matrix  
Default: [0;1;0] |
### patch

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>YData</td>
<td>The y-coordinates of the vertices of the patch</td>
<td>Value: vector or matrix</td>
</tr>
<tr>
<td></td>
<td>默认: [1;1;0]</td>
<td></td>
</tr>
<tr>
<td>ZData</td>
<td>The z-coordinates of the vertices of the patch</td>
<td>Value: vector or matrix</td>
</tr>
<tr>
<td></td>
<td>默认: [] (empty matrix)</td>
<td></td>
</tr>
</tbody>
</table>

### Specifying Color

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CData</td>
<td>Color data for use with the XData/YData/ZData method</td>
<td>Value: scalar, vector, or matrix</td>
</tr>
<tr>
<td></td>
<td>默认: [] (empty matrix)</td>
<td></td>
</tr>
<tr>
<td>CDataMapping</td>
<td>Controls mapping of CData to colormap</td>
<td>Values: scaled, direct</td>
</tr>
<tr>
<td></td>
<td>默认: scaled</td>
<td></td>
</tr>
<tr>
<td>EdgeColor</td>
<td>Color of face edges</td>
<td>Values: ColorSpec, none, flat, interp</td>
</tr>
<tr>
<td></td>
<td>默认: ColorSpec</td>
<td></td>
</tr>
<tr>
<td>FaceColor</td>
<td>Color of face</td>
<td>Values: ColorSpec, none, flat, interp</td>
</tr>
<tr>
<td></td>
<td>默认: ColorSpec</td>
<td></td>
</tr>
<tr>
<td>FaceVertexCData</td>
<td>Color data for use with Faces/Vertices method</td>
<td>Value: matrix</td>
</tr>
<tr>
<td></td>
<td>默认: [] (empty matrix)</td>
<td></td>
</tr>
<tr>
<td>MarkerEdgeColor</td>
<td>Color of marker or the edge color for filled markers</td>
<td>Values: ColorSpec, none, auto</td>
</tr>
<tr>
<td></td>
<td>默认: auto</td>
<td></td>
</tr>
<tr>
<td>MarkerFaceColor</td>
<td>Fill color for markers that are closed shapes</td>
<td>Values: ColorSpec, none, auto</td>
</tr>
<tr>
<td></td>
<td>默认: none</td>
<td></td>
</tr>
</tbody>
</table>

### Controlling the Effects of Lights

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmbientStrength</td>
<td>Intensity of the ambient light</td>
<td>Value: scalar &gt;=0 and &lt;=1</td>
</tr>
<tr>
<td></td>
<td>默认: 0.3</td>
<td></td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Property Value</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| BackFaceLighting            | Controls lighting of faces pointing away from camera       | Values: unlit, lit, reverselit  
Default: reverselit                             |
| DiffuseStrength             | Intensity of diffuse light                                | Value: scalar $\geq 0$ and $\leq 1$  
Default: 0.6                        |
| EdgeLighting                | Method used to light edges                                | Values: none, flat, gouraud, phong  
Default: none                           |
| FaceLighting                | Method used to light edges                                | Values: none, flat, gouraud, phong  
Default: none                           |
| NormalMode                  | MATLAB generated or user-specified normal vectors         | Values: auto, manual  
Default: auto                               |
| SpecularColorReflectance    | Composite color of specularly reflected light             | Value: scalar $0$ to $1$  
Default: 1                               |
| SpecularExponent            | Harshness of specular reflection                          | Value: scalar $\geq 1$  
Default: 10                             |
| SpecularStrength            | Intensity of specular light                               | Value: scalar $\geq 0$ and $\leq 1$  
Default: 0.9                             |
| VertexNormals               | Vertex normal vectors                                     | Value: matrix                                                               |

**Defining Edges and Markers**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| LineStyle           | Select from five line styles.           | Values: –, ––, :, –., none  
Default: –                         |
| LineWidth           | The width of the edge in points         | Value: scalar  
Default: 0.5 points             |
| Marker              | Marker symbol to plot at data points    | Values: see Marker property  
Default: none                           |
### patch

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MarkerSize</td>
<td>Size of marker in points</td>
<td>Value: size in points Default: 6</td>
</tr>
</tbody>
</table>

**Specifying Transparency**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaDataMapping</td>
<td>Transparency mapping method</td>
<td>Values: none, direct, scaled Default: scaled</td>
</tr>
<tr>
<td>EdgeAlpha</td>
<td>Transparency of the edges of patch faces</td>
<td>Values: scalar, flat, interp Default: 1 (opaque)</td>
</tr>
<tr>
<td>FaceAlpha</td>
<td>Transparency of the patch face</td>
<td>Values: scalar, flat, interp Default: 1 (opaque)</td>
</tr>
<tr>
<td>FaceVertexAlphaData</td>
<td>Face and vertex transparency data</td>
<td>Value: m-by-1 matrix</td>
</tr>
</tbody>
</table>

**Controlling the Appearance**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clipping</td>
<td>Clipping to axes rectangle</td>
<td>Values: on, off Default: on</td>
</tr>
<tr>
<td>EraseMode</td>
<td>Method of drawing and erasing the patch (useful for animation)</td>
<td>Values: normal, none, xor, background Default: normal</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Highlights patch when selected (Selected property set to on)</td>
<td>Values: on, off Default: on</td>
</tr>
<tr>
<td>Visible</td>
<td>Makes the patch visible or invisible</td>
<td>Values: on, off Default: on</td>
</tr>
</tbody>
</table>

**Controlling Access to Objects**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandleVisibility</td>
<td>Determines if and when the patch’s handle is visible to other functions</td>
<td>Values: on, callback, off Default: on</td>
</tr>
</tbody>
</table>
### Property Name | Property Description | Property Value
---|---|---
HitTest | Determines if the patch can become the current object (see the figure CurrentObject property) | Values: on, off Default: on

**Controlling Callback Routine Execution**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeingDeleted</td>
<td>Query to see if object is being deleted.</td>
<td>Values: on</td>
</tr>
<tr>
<td>BusyAction</td>
<td>Specifies how to handle callback routine interruption</td>
<td>Values: cancel, queue Default: queue</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Defines a callback routine that executes when a mouse button is pressed on over the patch</td>
<td>Value: string or function handle Default: '' (empty string)</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Defines a callback routine that executes when a patch is created</td>
<td>Value: string or function handle Default: '' (empty string)</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Defines a callback routine that executes when the patch is deleted (via close or delete)</td>
<td>Value: string or function handle Default: '' (empty string)</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Determines if callback routine can be interrupted</td>
<td>Values: on, off Default: on (can be interrupted)</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Associates a context menu with the patch</td>
<td>Value: handle of a Uicontextmenu</td>
</tr>
</tbody>
</table>

**General Information About the Patch**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Children</td>
<td>Patch objects have no children.</td>
<td>Value: [] (empty matrix)</td>
</tr>
<tr>
<td>Parent</td>
<td>The parent of a patch object is an axes, hggroup, or hgtransform object.</td>
<td>Value: object handle</td>
</tr>
</tbody>
</table>
### patch

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| Selected      | Indicates whether the patch is in a selected state | Values: on, off  
               | Default: on         |                |
| Tag           | User-specified label | Value: any string  
               | Default: '' (empty string) | |
| Type          | The type of graphics object (read only) | Value: the string 'patch' | |
| UserData      | User-specified data  | Value: any matrix  
               | Default: [] (empty matrix)   | |

**See Also**

area, caxis, fill, fill3, isosurface, surface
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 Objects for general information about this type of object.

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

**AlphaDataMapping** none | direct | {scaled}

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

- **none** — The transparency values of FaceVertexAlphaData are between 0 and 1 or are clamped to this range (the default).
- **scaled** — Transform the FaceVertexAlphaData to span the portion of the alphamap indicated by the axes ALim property, linearly mapping data values to alpha values.
- **direct** — Use the FaceVertexAlphaData as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from 1 to length(alphamap). MATLAB maps values less than 1 to the first alpha value in the alphamap, and values greater than length(alphamap) to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If FaceVertexAlphaData is an array of uint8 integers, then the indexing begins at 0 (i.e., MATLAB maps a value of 0 to the first alpha value in the alphamap).

**AmbientStrength** scalar >= 0 and <= 1

*Strength of ambient light.* This property sets the strength of the ambient light, which is a non-directional 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.
Patch Properties

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.
 Patch Properties

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

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.
Patch Properties

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.
Patch Properties

colormap. Values with a decimal portion are fixed to the nearest lower integer.

Children    matrix of handles
Always the empty matrix; patch objects have no children.

Clipping    {on} | off
Clipping to axes rectangle. When Clipping is on, MATLAB does not display any portion of the patch outside the axes rectangle.

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

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

```matlab
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 $\geq 0$ and $\leq 1$

*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:
Patch Properties

- **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.
Patch Properties

- **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.
Patch Properties

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

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:
Patch Properties

- 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` are set appropriately. The interpretation of the values specified for `FaceVertexCData` depends on the dimensions of the data.

For indexed colors, `FaceVertexCData` can be

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

For true colors, `FaceVertexCData` can be

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

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

HandleVisibility  {on}  |  callback  |  off

Control access to object’s handle by command-line users and GUIs. This property determines when an object’s handle is visible in its parent’s list of children. HandleVisibility is useful for preventing command-line users from accidentally 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
Patch Properties

protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

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

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

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

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

**HitTest**

{on} | off

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

**Interruptible**

{on} | off

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

**LineStyle**

{-} | — | : | —— | none

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

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

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

**LineWidth**

Scalar

*Edge line width.* The width, in points, of the patch edges (1 point = 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

**Marker Specifier** | **Description**
--- | ---
"d" | Diamond
"^" | 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} | 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 1/3 of the specified size.
NormalMode \{auto\} \mid 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 \mid \{off\}

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

SelectionHighlight \{on\} \mid off

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

\begin{itemize}
  \item Drawing handles at each vertex for a single-faced patch
  \item Drawing a dashed bounding box for a multifaced patch
\end{itemize}

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

SpecularColorReflectance scalar in the range 0 to 1

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

**SpecularExponent** scalar >= 1

*Harshness of specular reflection.* This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

**SpecularStrength** scalar >= 0 and <= 1

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

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

**Tag** string

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

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

```matlab
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.

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

**Type** string (read only)

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

**UIContextMenu** handle of a uicontextmenu object

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

**UserData**

*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**

*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**

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

**Visible**

*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**

*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**

*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**

*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 the MATLAB directory search path

**Graphical Interface**
As an alternative to the path function, use the **Set Path** dialog box. To open it, select **Set Path** from the **File** menu in the MATLAB desktop.

**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')` appends the `newpath` directory to the current search path.

`path('newpath', path)` prepends the `newpath` directory to the current search 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 in 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 rehash or Toolbox Path Caching.

**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, reloadedefaultpath, rmpath, savepath, startup, what

Search Path
**Path2rc**

**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.
pathdef

**Purpose**
List of directories in the MATLAB search path

**Tropical Interface**
As an alternative to using the `pathdef.m` file directly, use the **Set Path** dialog box. To open it, select **Set Path** from the **File** menu in the MATLAB desktop.

**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`.

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`

Search Path documentation, including:

- “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”
pathsep

Purpose
Return 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:

```
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

**Graphical Interface**
As an alternative to the `pathtool` function, select **Set Path** from the **File** menu 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.
pathtool

See Also

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

Search Path documentation, including, “Setting the Search Path”
Purpose

Halt execution temporarily

Syntax

```plaintext
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.

`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.

See Also

drawnow
Purpose
Set or query the plot box aspect ratio

Syntax
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 and the “Aspect Ratio” section in the Using MATLAB Graphics manual for a discussion of stretch-to-fill.

**Examples**

The following surface plot of the function \( z = xe^{(−x^2 − y^5)} \) 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.

\[ \text{pbaspect} \]

\[ \text{ans} = \]

---

2-1643
It is also interesting to look at the data aspect ratio selected by MATLAB.

```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.

```matlab
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.

```matlab
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

The “Aspect Ratio” section in the Using MATLAB Graphics manual
**Purpose**

Preconditioned Conjugate Gradients method

**Syntax**

\[
\begin{align*}
x &= \text{pcg}(A,b) \\
\text{pcg}(A,b,\text{tol}) & \\
\text{pcg}(A,b,\text{tol,\text{maxit}}) & \\
\text{pcg}(A,b,\text{tol,\text{maxit}},M) & \\
\text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2) & \\
\text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2,x0) & \\
\text{pcg}(A,b,\text{tol},M1,M2,x0,p1,p2,...) & \\
[x,\text{flag}] &= \text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2,x0,p1,p2,...) \\
[x,\text{flag,relres}] &= \text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2,x0,p1,p2,...) \\
[x,\text{flag,relres,iter}] &= \text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2,x0,p1,p2,...) \\
[x,\text{flag,relres,iter,\text{resvec}}] &= \text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2,x0,p1,p2,...)
\end{align*}
\]

**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 \( \text{afun} \) such that \( \text{afun}(x) \) returns \( A*x \).

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

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

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

\( \text{pcg}(A,b,\text{tol,\text{maxit}},M) \) and \( \text{pcg}(A,b,\text{tol,\text{maxit}},M1,M2) \) use symmetric positive definite preconditioner \( M \) or \( M = M1*M2 \) and effectively solve the system \( \text{inv}(M)*A*x = \text{inv}(M)*b \) for \( x \). If \( M \) is \( [] \) then pcg applies no preconditioner. \( M \) can be a function that 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.
pcg(afun, b, tol, maxit, m1fun, m2fun, x0, p1, p2, ...) passes parameters p1, p2, ... to functions afun(x, p1, p2, ...), m1fun(x, p1, p2, ...), and m2fun(x, p1, p2, ...).

[x, flag] = pcg(A, b, tol, maxit, M1, M2, x0) 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 tol within maxit iterations.</td>
</tr>
<tr>
<td>1</td>
<td>pcg 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>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 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] = pcg(A, b, tol, maxit, M1, M2, x0) also returns the relative residual norm(b - A*x)/norm(b). If flag is 0, relres <= tol.

[x, flag, relres, iter] = pcg(A, b, tol, maxit, M1, M2, x0) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

[x, flag, relres, iter, resvec] = pcg(A, b, tol, maxit, M1, M2, x0) also returns a vector of the residual norms at each iteration including norm(b - A*x0).

Examples

Example 1.

A = gallery('wilk', 21);
b = sum(A, 2);
tol = 1e-12;
maxit = 15;
M = diag([10:-1:1 1:10]);
[x,flag,rr,iter,rv] = pcg(A,b,tol,maxit,M);

Alternatively, use this one-line matrix-vector product function 

```matlab
function y = afun(x,n)
y = [0; x(1:n-1)] + [(1:(n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + [x(2:n); 0];
```

and this one-line preconditioner backsolve function 

```matlab
function y = mfun(r,n)
y = r ./ [(1:(n-1)/2:1)'; 1; (1:(n-1)/2)'];
```

as inputs to `pcg`

```
[x1,flag1,rr1,iter1,rv1] = pcg(@afun,b,tol,maxit,@mfun,... [],[],21);
```

**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), \ (backslash)

References

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

**Syntax**

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

**Description**

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

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

- \(\text{spline}\) produces a smoother result, i.e. \(S''(x)\) is continuous.
- \(\text{spline}\) produces a more accurate result if the data consists of values of a smooth function.
pchip

- 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


2-1652
**pcode**

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

**Syntax**

```plaintext
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.
**pcolor**

**Purpose**

Pseudocolor plot

**Syntax**

```
pcolor(C)
pcolor(X,Y,C)
pcolor(axes_handle,...)
h = pcolor(...)
```

**Description**

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

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

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

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

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

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

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

**Remarks**

A pseudocolor plot is a flat surface plot viewed from above. \( pcolor(X,Y,C) \) is the same as viewing \( surf(X,Y,0*Z,C) \) using \( view([0 90]) \).
When you use shading faceted or shading flat, the constant color of each cell is the color associated with the corner having the smallest $x$-$y$ coordinates. Therefore, $C(i,j)$ determines the color of the cell in the $i$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.

```matlab
pcolor(hadamard(20))
colormap(gray(2))
axis ij
axis square
```

A simple color wheel illustrates a polar coordinate system.

```matlab
n = 6;
r = (0:n)'/n;
theta = pi*(n:n)/n;
X = r*cos(theta);
Y = r*sin(theta);
C = r*cos(2*theta);
pcolor(X,Y,C)
```
Algorithm
The number of vertex colors for \texttt{pcolor(C)} is the same as the number of cells for \texttt{image(C)}. \texttt{pcolor} differs from \texttt{image} in that \texttt{pcolor(C)} specifies the colors of vertices, which are scaled to fit the colormap; changing the axes \texttt{clim} property changes this color mapping. \texttt{image(C)} specifies the colors of cells and directly indexes into the colormap without scaling. Additionally, \texttt{pcolor(X,Y,C)} can produce parametric grids, which is not possible with \texttt{image}.

See Also
\texttt{caxis, image, mesh, shading, surf, view}
pdepe

Purpose
Solve initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs) in one space variable and time.

Syntax
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options,p1,p2...)

Arguments
m
A parameter corresponding to the symmetry of the problem. m can be slab = 0, cylindrical = 1, or spherical = 2.
pdefun
A function that defines the components of the PDE.
icfun
A function that defines the initial conditions.
bcfun
A function that defines the boundary conditions.
xmesh
A vector [x0, x1, ..., xn] specifying the points at which a numerical solution is requested for every value in tspan. The elements of xmesh must satisfy x0 < x1 < ... < xn. The length of xmesh must be >= 3.
tspan
A vector [t0, t1, ..., tf] specifying the points at which a solution is requested for every value in xmesh. The elements of tspan must satisfy t0 < t1 < ... < tf. The length of tspan must be >= 3.
options
Some options of the underlying ODE solver are available in pdepe: RelTol, AbsTol, NormControl, InitialStep, and MaxStep. In most cases, default values for these options provide satisfactory solutions. See odeset for details.
p1,p2,...
Optional parameters to be passed to pdefun, icfun, and bcfun.

Description
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan) solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t. 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.
pdepe solves PDEs of the form:
\[
c(x, t, u, \frac{\partial u}{\partial x}) \frac{\partial u}{\partial t} = x^{-m} \frac{\partial}{\partial x} \left( x^m f(x, t, u, \frac{\partial u}{\partial x}) \right) + s(x, t, u, \frac{\partial u}{\partial x})
\]

(2-1)

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 2-1, \( f(x, t, u, \partial u/\partial x) \) is a flux term and \( s(x, t, u, \partial u/\partial x) \) is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix \( c(x, t, u, \partial u/\partial x) \). 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
\[
u(x, t_0) = u_0(x)
\]

(2-2)

For all \( t \) and either \( x = a \) or \( x = b \), the solution components satisfy a boundary condition of the form
\[
p(x, t, u) + q(x, t) f(x, t, u, \frac{\partial u}{\partial x}) = 0
\]

(2-3)

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 \( \partial u/\partial x \). Also, of the two coefficients, only \( p \) can depend on \( u \).

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

- \( m \) corresponds to \( m \).
- \( \text{xmesh}(1) \) and \( \text{xmesh}(\text{end}) \) correspond to \( a \) and \( b \).
- \( \text{tspan}(1) \) and \( \text{tspan}(\text{end}) \) correspond to \( t_0 \) and \( t_f \).
pdefun computes the terms \( c, f, \) and \( s \) (Equation 2-1). It has the form
\[
[c, f, s] = \text{pdefun}(x, t, u, dudx)
\]
The input arguments are scalars \( x \) and \( t \) and vectors \( u \) and \( dudx \) 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 2-1).

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 \( u \).

bcfun evaluates the terms \( p \) and \( q \) of the boundary conditions (Equation 2-3). It has the form
\[
[p_1, q_1, p_r, q_r] = \text{bcfun}(x_l, u_l, x_r, u_r, t)
\]
\( u_l \) is the approximate solution at the left boundary \( x_l = a \) and \( u_r \) is the approximate solution at the right boundary \( x_r = b \). \( p_1 \) and \( q_1 \) 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 \( a = 0 \). \text{pdepe} imposes this boundary condition automatically and it ignores values returned in \( p_1 \) and \( q_1 \).

\text{pdepe} returns the solution as a multidimensional array \( \text{sol} \).
\( u_i = u_i = \text{sol}(:,:,i) \) 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 \((t,x) = (\text{tspan}(j),\text{xmesh}(k))\).
\( u_i = \text{sol}(j,:,i) \) approximates component \( i \) of the solution at time \( \text{tspan}(j) \) and mesh points \( \text{xmesh}(:) \). Use \text{pdeval} to compute the approximation and its partial derivative \( \partial u_i / \partial x \) at points not included in \( \text{xmesh} \). See \text{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 \text{odeset} function. Only some of the options of the underlying ODE solver are available in \text{pdepe}: RelTol, AbsTol, NormControl,
InitialStep, and MaxStep. The defaults obtained by leaving off the input argument options will generally be satisfactory. See odeset for details.

sol = pdepe(m, pdefun, icfun, bcfun, xmesh, tspan, options, p1, p2...) passes the additional parameters p1, p2, ... to the functions pdefun, icfun, and bcfun. Use options = [] as a placeholder if no options are set.

**Remarks**

- The arrays xmesh and tspan play different roles in pdepe.
  - **tspan** The pdepe function performs the time integration with an ODE solver that selects both the time step and formula dynamically. The elements of tspan merely specify where you want answers and the cost depends weakly on the length of tspan.
  - **xmesh** Second order approximations to the solution are made on the mesh specified in xmesh. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. pdepe does not select the mesh in x automatically. You must provide an appropriate fixed mesh in xmesh. The cost depends strongly on the length of xmesh. When \( m > 0 \), it is not necessary to use a fine mesh near \( x = 0 \) to account for the coordinate singularity.

- The time integration is done with ode15s. pdepe exploits the capabilities of ode15s for solving the differential-algebraic equations that arise when Equation 2-1 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.

- After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not “consistent” with the discretization, pdepe tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, pdepe can find consistent initial conditions close to the given ones. If pdepe displays a message that it has difficulty finding consistent initial conditions, try refining the mesh.

No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.
Example 1. This example illustrates the straightforward formulation, computation, and plotting of the solution of a single PDE.

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

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')
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 * 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

\[
\begin{align*}
\frac{\partial u_1}{\partial t} &= 0.024 \frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2) \\
\frac{\partial u_2}{\partial t} &= 0.170 \frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)
\end{align*}
\]

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

\[
\begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \frac{\partial}{\partial t} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x} \begin{bmatrix} 0.024(\partial u_1 / \partial x) \\ 0.170(\partial u_2 / \partial x) \end{bmatrix} + \begin{bmatrix} -F(u_1 - u_2) \\ F(u_1 - u_2) \end{bmatrix}
\]

The boundary conditions on the partial derivatives of \( u \) 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

\[
\begin{bmatrix} 0 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 0.024(\partial u_1 / \partial x) \\ 0.170(\partial u_2 / \partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

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.
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(5.73*y)-exp(-11.47*y);
s = [-F; F];
% --------------------------------------------------------------
function u0 = pdex4ic(x);
u0 = [1; 0];
% --------------------------------------------------------------
function [pl,q1,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
q1 = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];

In this example, the PDEs, initial conditions, and boundary conditions are
coded in subfunctions pdex4pde, pdex4ic, and pdex4bc.
The surface plots show the behavior of the solution components.
See Also

function_handle, pdeval, ode15s, odeset, odeget

References

**pdeval**

**Purpose**
Evaluate the numerical solution of a PDE using the output of pdepe

**Syntax**
```matlab
[uout,duoutdx] = pdeval(m,xmesh,ui,xout)
```

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

**Description**
`[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution `ui` and its partial derivative `∂ui/∂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 `∂ui/∂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

A sample function of two variables.

Syntax

\[ Z = \text{peaks}; \]
\[ Z = \text{peaks}(n); \]
\[ Z = \text{peaks}(V); \]
\[ Z = \text{peaks}(X,Y); \]

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

\[ [X,Y,Z] = \text{peaks}; \]
\[ [X,Y,Z] = \text{peaks}(n); \]
\[ [X,Y,Z] = \text{peaks}(V); \]

Description

\text{peaks} is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating \text{mesh}, \text{surf}, \text{pcolor}, \text{contour}, and so on.

\[ Z = \text{peaks}; \] returns a 49-by-49 matrix.

\[ Z = \text{peaks}(n); \] returns an \( n \times n \) matrix.

\[ Z = \text{peaks}(V); \] returns an \( n \times n \) matrix, where \( n = \text{length}(V) \).

\[ Z = \text{peaks}(X,Y); \] evaluates \text{peaks} at the given \( X \) and \( Y \) (which must be the same size) and returns a matrix the same size.

\text{peaks}(\ldots) \ (\text{with no output argument}) \ plots \ the \ text{peaks} \ function \ with \ \text{surf}.

\[ [X,Y,Z] = \text{peaks}(\ldots); \] returns two additional matrices, \( X \) and \( Y \), for parametric plots, for example, \text{surf}(X,Y,Z,\text{del2}(Z)). \ If \ not \ given \ as \ input, \ the \ underlying \ matrices \ X \ and \ Y \ are

\[ [X,Y] = \text{meshgrid}(V,V) \]

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

\text{meshgrid}, \text{surf}
Perl

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, so MATLAB users can run M-files containing the perl function.

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

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

run the following statement in MATLAB

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

MATLAB returns

```matlab
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
nchoosek, permute, randperm
permute

**Purpose**
Rearrange the dimensions of a multidimensional 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:

```matlab
A = [1 2; 3 4]; permute(A,[2 1])
ans =
   1   3
   2   4
```

The following code permutes a three-dimensional array:

```matlab
X = rand(12,13,14); Y = permute(X,[2 3 1]); size(Y)
ans =
   13   14   12
```

**See Also**
ipermute
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).

See Also
clear, global, mislocked, mlock, munlock
pi

**Purpose**
Ratio of a circle's circumference to its diameter, $\pi$

**Syntax**
pi

**Description**
pi returns the floating-point number nearest the value of $\pi$. The expressions $4*\text{atan}(1)$ and $\text{imag}(\log(-1))$ provide the same value.

**Examples**
The expression $\sin(pi)$ is not exactly zero because pi is not exactly $\pi$.

```matlab
sin(pi)
```

```matlab
ans =
1.2246e-16
```

**See Also**
ans, eps, i, Inf, j, NaN
**Purpose**

Pie chart

**Syntax**

```
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,

```
pie(1:3,{'Taxes','Expenses','Profit'})
```

`pie(axes_handle,...)` plots into the axes with handle `axes_handle` instead of 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/sum(X)` to determine the area of each slice of the pie. If `sum(X) ≤ 1`, the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if `sum(X) < 1`.

**Examples**

Emphasize the second slice in the chart by setting its corresponding `explode` element to 1.

```
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie(x,explode)
colormap jet
```
See Also  pie3
**Purpose**
Three-dimensional pie chart

**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 handle axes_handle instead of the current axes (gca).

h = pie(...) 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:

```
x = [1 3 0.5 2.5 2]
explode = [0 1 0 0 0]
pie3(x,explode)
colormap hsv
```
**Purpose**

Moore-Penrose pseudoinverse of a matrix

**Syntax**

\[
B = \text{pinv}(A) \\
B = \text{pinv}(A,tol)
\]

**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 \( tol \) 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, \( \max(\text{size}(A)) \times \text{norm}(A) \times \text{eps} \).

**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

\[
\text{minimize} \ \|A*x-b\|
\]

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 = 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

\[
x = \\
\begin{bmatrix}
1.1538 \\
1.4615 \\
1.3846 \\
1.3846 \\
1.4615 \\
1.1538 \\
\end{bmatrix}
\]
and

\[ y = A\backslash b \]

which produces this result.

```
Warning: Rank deficient, rank = 3  tol =  1.8829e-013.
y =
    4.0000
    5.0000
    0
    0
    0
-1.0000
```

Both of these are exact solutions in the sense that \( \text{norm}(A \times x - b) \) and \( \text{norm}(A \times y - b) \) are on the order of roundoff error. The solution \( x \) is special because

\[ \text{norm}(x) = 3.2817 \]

is smaller than the norm of any other solution, including

\[ \text{norm}(y) = 6.4807 \]

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**
\[
\begin{align*}
x &= \begin{bmatrix} 3 \\ 4 \end{bmatrix}; \\
[G,y] &= \text{planerot}(x')
\end{align*}
\]

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

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

**See Also**
qrdelete, qrinsert
**Purpose**
Run published M-file demo

**Syntax**
playshow demoname

**Description**
playshow runs the published M-file demo `demoname`. To determine if a demo is a published M-file type, view the H1 line for the demo M-file, that is, the first comment line. If it begins with two comment symbols (%%), it is a published M-file demo.

**Examples**
The first line in `nesteddemo` begins with two comment symbols:

```matlab
%% Nested Function Examples
```

Therefore, type `playshow nesteddemo` to run the demo.

**See Also**
demo, helpbrowser
**plot**

**Purpose**
Linear 2-D plot

**Syntax**

```
plot(Y)
pplot(X1,Y1,...)
pplot(X1,Y1,LineSpec,...)
pplot(...,'PropertyName',PropertyValue,...)
pplot(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.

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” section for examples.)

plot(axes_handle,...) plots into the axes with handle axes_handle instead of 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 changes you make to these properties to persist, then you must define these changes as default values. For example,

```matlab
set(0,'DefaultAxesColorOrder',[0 0 0],...
    'DefaultAxesLineStyleOrder','-|-.|--|:');
```

sets the default `ColorOrder` to use only the color black and sets the `LineStyleOrder` to use solid, dash-dot, dash-dash, and dotted line styles.

Prevent Resetting of Color and Styles with `hold all`

The `all` option to the `hold` command prevents the `ColorOrder` and `LineStyleOrder` from being reset in subsequent `plot` commands. In the following sequence of commands, MATLAB continues to cycle through the colors defined by the axes `ColorOrder` property (see above).

```matlab
plot(rand(12,2))
hold all
plot(randn(12,2))
```

Additional Information

- See Creating Line Plots and Annotating Graphs for more information on plotting.
- See `LineSpec` for more information on specifying line styles and colors.
Examples

Specifying the Color and Size of Markers

You can also specify other line characteristics using graphics properties (see `line` for a description of these properties):

- **LineWidth** — Specifies the width (in points) of the line.
- **MarkerEdgeColor** — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- **MarkerFaceColor** — Specifies the color of the face of filled markers.
- **MarkerSize** — Specifies the size of the marker in units of points.

For example, these statements,

```matlab
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,...
     'MarkerEdgeColor','k',...
     'MarkerFaceColor','g',...
     'MarkerSize',10)
```

produce this graph.
Specifying Tick-Mark Location and Labeling

You can adjust the axis tick-mark locations and the labels appearing at each tick. For example, this plot of the sine function relabels the $x$-axis with more meaningful values:

```matlab
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

Now add axis labels and annotate the point $-\pi/4$, $\sin(-\pi/4)$.
**Adding Titles, Axis Labels, and Annotations**

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)',...
'HorizontalAlignment','left')
```

Now change the line color to red by first finding the handle of the line object created by `plot` and then setting its `Color` property. In the same statement, set the `LineWidth` property to 2 points.

```matlab
set(findobj(gca,'Type','line','Color',[0 0 1]),...
'Color','red',...
'LineWidth',2)
```
See Also

axis, bar, grid, hold, legend, line, LineSpec, loglog, plot3, plotyy, semilogx, semilogy, subplot, title, xlabel, xlim, ylabel, ylim, zlabel, zlim, stem

See the text String property for a list of symbols and how to display them.

See the Plot Editor for information on plot annotation tools in the figure window toolbar.

See Basic Plots and Graphs for related functions.
plot3

Purpose
3-D line plot

Syntax
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)

Description
The `plot3` function displays a three-dimensional plot of a set of data points.

`plot3(X1,Y1,Z1,...)`, where `X1`, `Y1`, `Z1` are vectors or matrices, plots one or more lines in three-dimensional space through the points whose coordinates are the elements of `X1`, `Y1`, and `Z1`.

`plot3(X1,Y1,Z1,LineSpec,...)` creates and displays all lines defined by the `Xn,Yn,Zn,LineSpec` quads, where `LineSpec` is a line specification that determines line style, marker symbol, and color of the plotted lines.

`plot3(...,'PropertyName',PropertyValue,...)` sets properties to the specified property values for all line graphics objects created by `plot3`.

`h = plot3(...)` returns a column vector of handles to lineseries graphics objects, with one handle per object.

Remarks
If one or more of `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
```
See Also

axis, bar3, grid, line, LineSpec, loglog, plot, semilogx, semilogy, subplot
plotbrowser

**Purpose**
Show or hide figure plotbrowser

**Syntax**
plotbrowser('on')  
plotbrowser('off')  
plotbrowser('toggle')  
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**
figurepalette, propertyeditor
**Purpose**
Start plot edit mode to allow editing and annotation of plots

**Syntax**

```matlab
plotedit on
plotedit off
plotedit
plotedit('state')
plotedit(h)
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 edit 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 Tools menu in the menu bar</td>
</tr>
<tr>
<td>hidetoolsmenu</td>
<td>Removes the Tools 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`. 

2-1693
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 figure.

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:

    plotedit('hidetoolsmenu')

See Also

axes, line, open, plot, print, saveas, text, propedit
**Purpose**

Draw scatter plots

**Syntax**

```matlab
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 `'.`.

`[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
plottools

**Purpose**  
Show or hide the plot tools

**Syntax**  
plottools('on')  
plottools('off')  
plottools(figure_handle,...)  
plottools(...,'tool')

**Description**  
plottools('on') displays the Figure Palette, Plot Browser, and Property Editor on the current figure.

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 the current figure.

plottools(...,'tool') operates on the specified tool only. **tool** can be one of the following strings:

- figurepalette
- plotbrowser
- propertyeditor

**See Also**  
figurepalette, plotbrowser, propertyeditor
**Purpose**  
Create graphs with y-axes on both left and right side

**Syntax**

```matlab
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,'function')
plotyy(X1,Y1,X2,Y2,'function1','function2')
[AX,H1,H2] = plotyy(...)```

**Description**

`plotyy(X1,Y1,X2,Y2)` plots $X1$ versus $Y1$ with y-axis labeling on the left and plots $X2$ versus $Y2$ with y-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,'function')` uses the specified plotting function to produce the graph.

*function* can be either a function handle or a string specifying `plot`, `semilogx`, `semilogy`, `loglog`, `stem`, or any MATLAB function that accepts the syntax

```matlab
h = function(x,y)
```

For example,

```matlab
plotyy(x1,y1,x2,y2,@loglog) % function handle
plotyy(x1,y1,x2,y2,'loglog') % string
```

Function handles enable you to access user-defined subfunctions and can provide other advantages. See `@` for more information on using function handles.

`plotyy(X1,Y1,X2,Y2,'function1','function2')` uses `function1(X1,Y1)` to plot the data for the left axis and `function2(X2,Y2)` to plot the data for the right axis.

`[AX,H1,H2] = plotyy(...)` returns the handles of the two axes created in `AX` and the handles of the graphics objects from each plot in `H1` and `H2`. `AX(1)` is the left axes and `AX(2)` is the right axes.

**Examples**

This example graphs two mathematical functions using `plot` as the plotting function. The two y-axes enable you to display both sets of data on one graph even though relative values of the data are quite different.

```matlab
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
```
\[ y_2 = 0.8 \times \exp(-0.5 \times x) \times \sin(10 \times x); \]

\[ [AX,H1,H2] = \text{plotyy}(x,y1,x,y2,'plot'); \]

You can use the handles returned by \text{plotyy} to label the axes and set the line styles used for plotting. With the axes handles you can specify the \text{YLabel} properties of the left- and right-side \( y \)-axis:

\[
\begin{align*}
\text{set}(\text{get}(AX(1),'Ylabel')),'String','Left \ Y-axis') \\
\text{set}(\text{get}(AX(2),'Ylabel')),'String','Right \ Y-axis')
\end{align*}
\]

Use the \text{xlabel} and \text{title} commands to label the \( x \)-axis and add a title:

\[
\begin{align*}
\text{xlabel}('Zero \ to \ 20 \ \mu\text{sec}.') \\
\text{title}('Labeling \ \text{plotyy}')
\end{align*}
\]

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

\[
\begin{align*}
\text{set}(H1,'LineStyle','--') \\
\text{set}(H2,'LineStyle',':')
\end{align*}
\]
See Also

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

See Using Multiple X- and Y-Axes for more information.
**pol2cart**

**Purpose**
Transform polar or cylindrical coordinates to Cartesian

**Syntax**

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

**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.

\[X,Y,Z] = \text{pol2cart}(\text{THETA}, \text{RHO}, \text{Z})\] transforms the cylindrical coordinate data stored in corresponding elements of \(\text{THETA}\), \(\text{RHO}\), and \(\text{Z}\) to three-dimensional Cartesian, or \(xyz\), coordinates. The arrays \(\text{THETA}\), \(\text{RHO}\), and \(\text{Z}\) 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:

**Polar to Cartesian Mapping**

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

**Cylindrical to Cartesian Mapping**

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

**See Also**
cart2pol, cart2sph, sph2cart

2-1702
Purpose
Plot polar coordinates

Syntax
polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axes_handle,...)
h = polar(...)

Description
The polar function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

polar(theta,rho) creates a polar coordinate plot of the angle theta versus the radius rho. theta is the angle from the x-axis to the radius vector specified in radians; rho is the length of the radius vector specified in dataspace units.

polar(theta,rho,LineSpec) LineSpec specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

polar(axes_handle,...) plots into the axes with handle axes_handle instead of the current axes (gca).

h = polar(...) returns the handle of a line object in h.

Examples
Create a simple polar plot using a dashed red line:

t = 0:.01:2*pi;
polar(t,sin(2*t).*cos(2*t), '--r')
polar

See Also
cart2pol, compass, LineSpec, plot, pol2cart, rose
**Purpose**
Polynomial with specified roots

**Syntax**

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

**Description**

\[
p = \text{poly}(A) \quad \text{where} \quad A \text{ is an } n \times n \text{ matrix returns an } n+1 \text{ element row vector} \\
\text{whose elements are the coefficients of the characteristic polynomial,} \\
\det(sI - A). \text{ The coefficients are ordered in descending powers: if a vector } c \text{ has} \\
n+1 \text{ components, the polynomial it represents is } c_1 s^n + \ldots + c_n s + c_{n+1} \\
\]

\[
p = \text{poly}(r) \quad \text{where } r \text{ is a vector returns a row vector whose elements are the} \\
coefficients of the polynomial whose roots are the elements of } r.
\]

**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 \(A\)

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

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

\[
p = \text{poly}(A) \\
p = \\
\begin{bmatrix}
1 & -6 & -72 & -27
\end{bmatrix}
\]

The roots of this polynomial (eigenvalues of matrix \(A\)) are returned in a column vector by \(\text{roots}\):

\[
r = \text{roots}(p)
\]
The algorithms employed for poly and roots illustrate an interesting aspect of the modern approach to eigenvalue computation. poly(A) generates the characteristic polynomial of A, and roots(poly(A)) finds the roots of that polynomial, which are the eigenvalues of A. But both poly and roots use eig, which is based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If A is an n-by-n matrix, poly(A) produces the coefficients c(1) through c(n+1), with c(1) = 1, in

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

The algorithm is

\[ z = \text{eig}(A); \]
\[ c = \text{zeros}(n+1,1); c(1) = 1; \]
\[ \text{for } j = 1:n \]
\[ \quad c(2:j+1) = c(2:j+1) - z(j)*c(1:j); \]
\[ \text{end} \]

This recursion is easily derived by expanding the product.

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

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 = polyarea(X,Y)
A = polyarea(X,Y,dim)

**Description**

A = 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 polyarea returns the area of polygons defined by the columns X and Y.

If X and Y are multidimensional arrays, polyarea returns the area of the polygons in the first nonsingleton dimension of X and Y.

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

**Examples**

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

See Also

convhull, inpolygon, rectint
### polyder

**Purpose**
Polynomial derivative

**Syntax**
- `k = polyder(p)`
- `k = polyder(a,b)`
- `[q,d] = polyder(b,a)`

**Description**
The `polyder` function calculates the derivative of polynomials, polynomial products, and polynomial quotients. The operands `a`, `b`, and `p` are vectors whose elements are the coefficients of a polynomial in descending powers.

- `k = polyder(p)` returns the derivative of the polynomial `p`.
- `k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`.
- `[q,d] = polyder(b,a)` returns the numerator `q` and denominator `d` of the derivative of the polynomial quotient `b/a`.

**Examples**
The derivative of the product

\[(3x^2 + 6x + 9)(x^2 + 2x)\]

is obtained with

```matlab
a = [3 6 9];
b = [1 2 0];
k = polyder(a,b)
k =
12 36 42 18
```

This result represents the polynomial

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

**See Also**
`conv`, `deconv`

2-1708
Purpose
Polynomial eigenvalue problem

Syntax
\[ [X,e] = \text{polyeig}(A_0,A_1,...,A_p) \]
e = \text{polyeig}(A_0,A_1,...,A_p)
\[ [X, e, s] = \text{polyeig}(A_0,A_1,...,A_p) \]

Description
\[ [X,e] = \text{polyeig}(A_0,A_1,...,A_p) \]
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 \( n \)-by-\( n*p \), whose columns are the eigenvectors, and a vector \( e \), of length \( n*p \), 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.

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

\[ [X, e, s] = \text{polyeig}(A_0,A_1,...,A_p) \]
also returns a \( p*n \) length vector \( s \), of length \( p*n \), 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 \( \text{polyeig}(a_0,a_1,...,a_p) \) for scalars \( a_0, a_1, ..., a_p \) 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.
Algorithm
The `polyeig` function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of `eig` and `qz` for more on this.

See Also
`coneig`, `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

\[
p(x) = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_{n+1}
\]

\( [p,S] = \text{polyfit}(x,y,n) \) returns the polynomial coefficients \( p \) and a structure \( S \) for use with \( \text{polyval} \) to obtain error estimates or predictions. 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 =
0.0084  -0.0983  0.4217  -0.7435  0.1471  1.1064  0.0004

There are seven coefficients and the polynomial is

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

To see how good the fit is, evaluate the polynomial at the data points with

\[ f = \text{polyval}(p,x); \]

A table showing the data, fit, and error is

\[
\text{table} = [x \ y \ f \ y-f]
\]

\[
\text{table} =
\begin{array}{cccc}
0 & 0 & 0.0004 & -0.0004 \\
0.1000 & 0.1125 & 0.1119 & 0.0006 \\
0.2000 & 0.2227 & 0.2223 & 0.0004 \\
0.3000 & 0.3286 & 0.3287 & -0.0001 \\
0.4000 & 0.4284 & 0.4288 & -0.0004 \\
\ldots \\
2.1000 & 0.9970 & 0.9969 & 0.0001 \\
2.2000 & 0.9981 & 0.9982 & -0.0001 \\
2.3000 & 0.9989 & 0.9991 & -0.0003 \\
2.4000 & 0.9993 & 0.9995 & -0.0002 \\
2.5000 & 0.9996 & 0.9994 & 0.0002 \\
\end{array}
\]

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.

\[
x = (0: 0.1: 5)';
y = \text{erf}(x);
f = \text{polyval}(p,x);
\text{plot}(x,y,'o',x,f,'-')
\text{axis([0 5 0 2])}
\]
Algorithm

The `polyfit` M-file forms the Vandermonde matrix, $V$, whose elements are powers of $x$.

$$v_{i,j} = x_i^{n-j}$$

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

$$Vp \equiv y$$

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

See Also

`poly`, `polyval`, `roots`
**polyint**

**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 = polyval(p,x)
y = polyval(p,x,[],mu)
[y,delta] = polyval(p,x,S)
[y,delta] = polyval(p,x,S,mu)

Description  
y = polyval(p,x) returns the value of a polynomial of degree n evaluated at x. The input argument p is a vector of length n+1 whose elements are the coefficients in descending powers of the polynomial to be evaluated.

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

x can be a matrix or a vector. In either case, polyval evaluates p at each element of x.

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

Remarks  
The polyvalm(p,x) function, with x a matrix, evaluates the polynomial in a matrix sense. See polyvalm for more information.

Examples  
The polynomial \( p(x) = 3x^2 + 2x + 1 \) is evaluated at \( x = 5, 7, \) and 9 with

\[
p = [3 \ 2 \ 1];\n\text{polyval}(p,[5 \ 7 \ 9])\n\]

which results in

\[
\text{ans = }
\begin{bmatrix}
86 \\
162 \\
262
\end{bmatrix}
\]

For another example, see polyfit.
polyval

See Also   polyfit, polyvalm
**Purpose**  
Matrix polynomial evaluation

**Syntax**

\[ Y = \text{polyvalm}(p,X) \]

**Description**

\( Y = \text{polyvalm}(p,X) \) evaluates a polynomial in a matrix sense. This is the same as substituting matrix \( X \) in the polynomial \( p \).

Polynomial \( p \) is a vector whose elements are the coefficients of a polynomial in descending powers, and \( X \) must be a square matrix.

**Examples**

The Pascal matrices are formed from Pascal’s triangle of binomial coefficients. Here is the Pascal matrix of order 4.

\[
X = \text{pascal}(4) \\
X =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 20 \\
\end{bmatrix}
\]

Its characteristic polynomial can be generated with the \text{poly} function.

\[
p = \text{poly}(X) \\
p =
\begin{bmatrix}
1 & -29 & 72 & -29 & 1 \\
\end{bmatrix}
\]

This represents the polynomial \( x^4 - 29x^3 + 72x^2 - 29x + 1 \).

Pascal matrices have the curious property that the vector of coefficients of the characteristic polynomial is palindromic; it is the same forward and backward.

Evaluating this polynomial at each element is not very interesting.

\[
\text{polyval}(p,X) \\
\text{ans} =
\begin{bmatrix}
16 & 16 & 16 & 16 \\
16 & 15 & -140 & -563 \\
16 & -140 & -2549 & -12089 \\
16 & -563 & -12089 & -43779 \\
\end{bmatrix}
\]

But evaluating it in a matrix sense is interesting.

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

ans =
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

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 * 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 = 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
Evaluate piecewise polynomial.

Syntax

\[
\begin{align*}
v &= \text{ppval}(pp, xx) \\
v &= \text{ppval}(xx, pp)
\end{align*}
\]

Description

\[
v = \text{ppval}(pp, xx) \text{ returns the value at the points } xx \text{ of the piecewise polynomial contained in } pp, \text{ as constructed by } \text{spline} \text{ or the spline utility } \text{mkpp}.
\]

\[
v = \text{ppval}(xx, pp) \text{ returns the same result but can be used with functions like } \text{fminbnd}, \text{fzero and quad} \text{ that take a function as an argument.}
\]

Examples

Compare the results of integrating the function \( \cos \)

\[
a = 0; \ b = 10;
int1 = \text{quad}(\cos, a, b, [], [])
\]

\[
\begin{align*}int1 &= -0.5440
\end{align*}
\]

with the results of integrating the piecewise polynomial \( pp \) that approximates the cosine function by interpolating the computed values \( x \) and \( y \).

\[
x = a:b;
y = \cos(x);
pp = \text{spline}(x, y);
int2 = \text{quad}(\text{ppval}, a, b, [], [], pp)
\]

\[
\begin{align*}
int2 &= -0.5485
\end{align*}
\]

\( int1 \) 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
\( \text{mkpp}, \text{spline}, \text{unmkpp} \)
Purpose
Return directory containing preferences, history, and layout files

Syntax
```matlab
prefdir
d = prefdir
d = prefdir(1)
```

Description
`prefdir` returns the directory that contains preferences for MATLAB and related products (`matlab.prf`), the command history (`history.m`), the MATLAB shortcuts (`shortcuts.xml`), and the MATLAB desktop layout files (`MATLABDesktop.xml` and `Your_Saved_LayoutMATLABLayout.xml`).

On Macintosh platforms, the directory might be in a hidden folder, for example, `myname/.matlab/R14`. To access the directory, select `Go -> Go to Folder` in the Finder. In the resulting dialog box, type the path returned by `prefdir` and press `Enter`.

```matlab
d = prefdir
```
d returns the name of the directory containing preferences and related files, but does not ensure its existence.

```matlab
d = prefdir(1)
```
creates a directory for preferences and related files if one does not exist.

Examples
Run `prefdir`.
MATLAB returns
```matlab
ans =
C:\WINNT\Profiles\tbear.MATHWORKS\Application Data\MathWorks\MATLAB\R14
```
Running `dir` for the directory shows
```matlab
.              history.m
..             matlab.prf
cwdhistory.m   MATLABDesktop.xml
shortcuts.xml
```
and possibly other files for other MathWorks products and any desktop layouts you saved.
prefdir

See Also

Fonts, Colors, and Other Preferences
**Purpose** Display Preferences dialog box for MATLAB and related products

**Graphical Interface** As an alternative to the preferences function, select Preferences from the File menu 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. For more information, see Fonts, Colors, and Other Preferences.
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

\[ p = \text{primes}(37) \]

\[ p = \]

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

See Also

factor
Purpose

Create hardcopy output

Syntax

print
print filename
print -ddriver
print -dformat
print -dformat filename
print -smodelname
print ... -options
[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 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 -dformat 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 -smodelname 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, printopt**

`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><code>lpr -r</code></td>
<td><code>dps2</code></td>
</tr>
<tr>
<td>Windows</td>
<td><code>COPY /B %s LPT1:</code></td>
<td><code>dwin</code></td>
</tr>
</tbody>
</table>

**Drivers**

The table below shows the complete list of printer drivers supported by MATLAB. If you do not specify a driver, MATLAB uses the default setting shown in the previous table.

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><code>-dbj10e</code></td>
<td>Yes</td>
</tr>
<tr>
<td>Canon BubbleJet BJ200 color</td>
<td><code>-dbj200</code></td>
<td>Yes</td>
</tr>
<tr>
<td>Canon Color BubbleJet BJC-70/BJC-600/BJC-4000</td>
<td><code>-dbjc600</code></td>
<td>Yes</td>
</tr>
<tr>
<td>Canon Color BubbleJet BJC-800</td>
<td><code>-dbjc800</code></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>DEC LN03</td>
<td>-dln03</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</strong> and compatible; color (not supported on HP-700)</td>
<td>-depsonc</td>
<td>Yes</td>
</tr>
<tr>
<td>Fujitsu 3400/2400/1200</td>
<td>-depsonc</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DesignJet 650C color (not supported on Windows)</td>
<td>-ddnj650c</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet 500</td>
<td>-ddjet500</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet 500C (creates black and white output)</td>
<td>-dcdjmono</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet 500C (with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows)</td>
<td>-dcdjcolor</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet 500C/540C color (not supported on Windows)</td>
<td>-dcdj500</td>
<td>Yes</td>
</tr>
<tr>
<td>HP Deskjet 550C color (not supported on Windows)</td>
<td>-dcdj550</td>
<td>Yes</td>
</tr>
<tr>
<td>HP DeskJet and DeskJet Plus</td>
<td>-ddeskjet</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet</td>
<td>-dlaserjet</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet+</td>
<td>-dljetplus</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet IIIP</td>
<td>-dljet2p</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet III</td>
<td>-dljet3</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet 4.5L and 5P</td>
<td>-dljet4</td>
<td>Yes</td>
</tr>
<tr>
<td>HP LaserJet 5 and 6</td>
<td>-dpxlmono</td>
<td>Yes</td>
</tr>
</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.

---

**Printer Driver** | **PRINT Command Option String** | **GhostScript**
--- | --- | ---
**HP PaintJet color** | -dpaintjet | Yes
**HP PaintJet XL color** | -dpjxl | Yes
**HP PaintJet XL color** | -dpjetxl | Yes
**HP PaintJet XL300 color (not supported on Windows)** | -dpjxl300 | Yes
**HPGL for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)** | -dhpgl | No
**IBM 9-pin Proprinter** | -dibmprom | Yes
**PostScript black and white** | -dps | No
**PostScript color** | -dpsc | No
**PostScript Level 2 black and white** | -dps2 | No
**PostScript Level 2 color** | -dpasc2 | No
**Windows color (Windows only)** | -dwinc | No
**Windows monochrome (Windows only)** | -dwin | No

---

**Graphics Format Files**

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 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. The first column indicates this by showing “MATLAB” or “Ghostscript” in parentheses. 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>BMP monochrome BMP</td>
<td>Bitmap</td>
<td>-dbmpmono</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>BMP 24-bit BMP</td>
<td>Bitmap</td>
<td>-dbmp16m</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>BMP 8-bit (256-color) BMP *this format uses a fixed colormap</td>
<td>Bitmap</td>
<td>-dbmp256</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>BMP 24-bit</td>
<td>Bitmap</td>
<td>-dbmp</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EMF</td>
<td>Vector</td>
<td>-dmeta</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS black and white</td>
<td>Vector</td>
<td>-deps</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS color</td>
<td>Vector</td>
<td>-depsc</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS Level 2 black and white</td>
<td>Vector</td>
<td>-deps2</td>
<td>MATLAB</td>
</tr>
<tr>
<td>EPS Level 2 color</td>
<td>Vector</td>
<td>-depsc2</td>
<td>MATLAB</td>
</tr>
<tr>
<td>HDF 24-bit</td>
<td>Bitmap</td>
<td>-dhdf</td>
<td>MATLAB</td>
</tr>
<tr>
<td>ILL (Adobe Illustrator)</td>
<td>Vector</td>
<td>-dill</td>
<td>MATLAB</td>
</tr>
<tr>
<td>JPEG 24-bit</td>
<td>Bitmap</td>
<td>-djpege</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>-dpbraw</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX 1-bit</td>
<td>Bitmap</td>
<td>-dpcoxmono</td>
<td>Ghostscript</td>
</tr>
<tr>
<td>PCX 24-bit color PCX file format, three 8-bit planes</td>
<td>Bitmap</td>
<td>-dpcox24b</td>
<td>Ghostscript</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.

**Options**

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>-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 <strong>Print Setup</strong> 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 and Ghostscript only. Specify resolution in dots per inch. See “Setting the Resolution.”</td>
</tr>
</tbody>
</table>
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>
</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.
Printing Tips

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:

<table>
<thead>
<tr>
<th>Property Value</th>
<th>Size (Width-by-Height)</th>
</tr>
</thead>
<tbody>
<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>
• 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 \texttt{−dps}.

• The behavior you are experiencing may 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 \texttt{Edit→Copy Figure} menu item on the figure window menu or the \texttt{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 \texttt{File→Preferences→Copying Options} dialog box. The Windows Enhanced Metafile clipboard format produces a better quality image than Windows Bitmap.

\textbf{Printing MATLAB GUIs}

You can generally obtain better results when printing a figure window that contains MATLAB \texttt{uicontrols} by setting these key properties:

• Set the figure \texttt{PaperPositionMode} property to \texttt{auto}. This ensures the printed version is the same size as the onscreen version. With \texttt{PaperPositionMode} set to \texttt{auto} MATLAB does not resize the figure to fit the current value of the \texttt{PaperPosition}. This is particularly important if you have specified a figure \texttt{ResizeFcn}, because if MATLAB resizes the figure during the print operation, the \texttt{ResizeFcn} is automatically called.

To set \texttt{PaperPositionMode} on the current figure, use the command

\begin{verbatim}
set(gcf,'PaperPositionMode','auto')
\end{verbatim}

• Set the figure \texttt{InvertHardcopy} property to \texttt{off}. By default, MATLAB changes the figure background color of printed output to white, but does not change the color of \texttt{uicontrols}. If you have set the background color, for example, to match the gray of the GUI devices, you must set \texttt{InvertHardcopy} to \texttt{off} to preserve the color scheme.

To set \texttt{InvertHardcopy} on the current figure, use the command

\begin{verbatim}
set(gcf,'InvertHardcopy','off')
\end{verbatim}
• 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 may actually 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 No. 2”, its handle is 2. The syntax is
This example prints the figure whose handle is 2, regardless of which figure is the current figure.

```plaintext
print -f
```

**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.

```plaintext
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.

```plaintext
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,

```plaintext
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.

```plaintext
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`.

```plaintext
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.

\[
\text{print('Thruster Control')}
\]

To print the current system, use

\[
\text{print -s}
\]

For information about issues specific to printing Simulink windows, see the Simulink documentation.

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.

\[
\begin{align*}
\text{surf(peaks)} \\
\text{shading interp} \\
\text{set(gcf,'PaperPositionMode','auto')} \\
\text{print -dpsc2 -zbuffer -r200}
\end{align*}
\]

**Batch Processing**

You can use the function form of print to pass variables containing file names. For example, this for loop creates a series of graphs and prints each one with a different file name.

\[
\begin{align*}
\text{for } k=1: \text{length(fnames)} \\
\text{surf(Z(:,:,k))} \\
\text{print('-dtiff','-r200',fnames(k))}
\end{align*}
\]

**Tiff Preview**

The command

\[
\text{print -depsc -tiff -r300 picture1}
\]

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
printdlg

Purpose
Display print dialog box

Syntax
printdlg
printdlg(fig)
printdlg('-crossplatform',fig)
printdlg('-setup',fig)

Description
printdlg prints the current figure.

printdlg(fig) creates a 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.
printpreview

Purpose

Preview figure
to be printed

Syntax

printpreview
printpreview(f)

Description

printpreview displays a dialog box showing the figure in the currently active figure window as it will be printed. The figure is displayed with a 1/4 size thumbnail or full size image.

printpreview(f) displays a dialog box showing the figure having the handle f as it will be printed.

You can select any of the following options from the Print Preview dialog box.

<table>
<thead>
<tr>
<th>Option Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print...</td>
<td>Close Print Preview and open the Print dialog</td>
</tr>
<tr>
<td>Page Setup...</td>
<td>Open the Page Setup dialog</td>
</tr>
<tr>
<td>Zoom In</td>
<td>Display a full size image of the page</td>
</tr>
<tr>
<td>Zoom Out</td>
<td>Display a 1/4 scaled image of the page</td>
</tr>
<tr>
<td>Close</td>
<td>Close the Print Preview dialog</td>
</tr>
</tbody>
</table>

See Also

printdlg, pagesetupdlg
**Purpose**
Product of array elements

**Syntax**

B = prod(A)
B = prod(A,dim)

**Description**

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

If A is a vector, prod(A) returns the product of the elements.

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

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

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

**Examples**

The magic square of order 3 is

M = magic(3)

M =

<table>
<thead>
<tr>
<th>8</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

The product of the elements in each column is

prod(M) =

| 96 | 45 | 84 |

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

prod(M,2) =

<table>
<thead>
<tr>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
</tr>
<tr>
<td>72</td>
</tr>
</tbody>
</table>

**See Also**
cumprod, diff, sum
Profile the execution time for a function

As an alternative to the profile function, select Desktop -> Profiler from the desktop.

profile on
profile on -detail level
profile on -history
profile off
profile resume
profile clear
profile viewer
s = profile('status')
stats = profile('info')

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 on starts the Profiler, clearing previously recorded profile statistics.

profile on -detail level starts the Profiler, clearing previously recorded profile statistics, and specifying the set of functions you want to profile. Use the following text strings as the value of the -detail option, level.

<table>
<thead>
<tr>
<th>Value for level</th>
<th>Gathers Information About</th>
</tr>
</thead>
<tbody>
<tr>
<td>'builtin'</td>
<td>M-functions, M-subfunctions, and MEX-functions,</td>
</tr>
<tr>
<td></td>
<td>plus built-in functions, such as eig</td>
</tr>
<tr>
<td>'mmex'</td>
<td>M-functions, M-subfunctions, and MEX-functions.</td>
</tr>
<tr>
<td></td>
<td>This is the default value.</td>
</tr>
</tbody>
</table>

profile on -history starts the Profiler, clearing previously recorded profile statistics, and recording 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 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.

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>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 functions 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>
</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>Name</td>
<td>Name of the profiler</td>
</tr>
<tr>
<td>ClockSpeed</td>
<td>Estimated clock speed of the CPU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionName</td>
<td>Function name, includes subfunction references</td>
</tr>
<tr>
<td>FileName</td>
<td>Filename is a fully qualified path</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 this function was called</td>
</tr>
<tr>
<td>TotalTime</td>
<td>Total time spent in this function and its child functions</td>
</tr>
<tr>
<td>TotalRecursiveTime</td>
<td>No longer used. Ignore value.</td>
</tr>
<tr>
<td>Children</td>
<td><code>FunctionTable</code> indices to child functions</td>
</tr>
<tr>
<td>Parents</td>
<td><code>FunctionTable</code> indices to parent functions</td>
</tr>
</tbody>
</table>
Examples

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')
```

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)
```
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

See Profiling for Improving Performance
### 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.
```matlab
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

### See Also
- `profile`

Profiling for Improving Performance
propedit

**Purpose**
Starts the Property Editor

**Syntax**

propedit

propedit(handle_list)

propedit(handle_list,'v6')

**Description**
propedit starts the Property Editor, a graphical user interface to the properties of graphics objects. There must be a current figure to call propedit without an object handle.

propedit(handle_list) edits the properties for the object (or objects) in handle_list.

propedit(handle_list, 'v6') displays the MATLAB Version 6 Property Editor.

Starting the Property Editor enables plot editing mode for the figure.

---

**Note**
The Version 6 Property Editor may not work with all objects.

**See Also**
inspect, plotedit, propertyeditor
**Purpose**
Show or hide property editor

**Syntax**
- `propertyeditor('on')`
- `propertyeditor('off')`
- `propertyeditor('toggle')`, `propertyeditor`
- `propertyeditor(figure_handle,...)`

**Description**
- `propertyeditor('on')` displays the property editor on the current figure.
- `propertyeditor('off')` hides the property editor on the current figure.
- `propertyeditor('toggle')` or `propertyeditor` toggles the visibility of the property editor on the current figure.
- `propertyeditor(figure_handle,...)` displays or hides the property editor on the figure specified by `figure_handle`.

**See Also**
- `plotbrowser`, `figurepalette`
**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

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

\[ Y = \psi(k,X) \] evaluates the \(k\)th derivative of \(\psi\) at the elements of \(X\). \(\psi(0,X)\) is the digamma function, \(\psi(1,X)\) is the trigamma function, \(\psi(2,X)\) is the tetragamma function, etc.

\[ Y = \psi(k0:k1,X) \] evaluates derivatives of order \(k0\) through \(k1\) at \(X\). \(Y(k,j)\) is the \((k-1+k0)\)th derivative of \(\psi\), evaluated at \(X(j)\).

**Examples**

**Example 1.** Use the \(\psi\) function to calculate Euler's constant, \(\gamma\).

```
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 \((\pi^2/6) - 1\).

```
format long
psi(1,2)
ans =
 0.64493406684823
```
\[ \psi(0:1,x)' \times 1 \]

**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**

publish

Purpose
Run M-file containing cells, and save results to file of specified type

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

Syntax
publish('script')
publish('script', 'format')
publish('script', 'options')

Description
publish('script') runs the file named script and publishes the code, comments, and results to an HTML output file. The output file is named script.html and is stored, along with other supporting output files, in an html subdirectory in script's directory.

publish('script', 'format') runs the file named script and publishes the code, comments, and results to an output file using the specified format. Allowable values for format are html (the default), xml, tex for LaTeX, doc for Microsoft Word documents, and ppt for Microsoft PowerPoint documents. The output file is named script.format and is stored, along with other supporting output files, in an html subdirectory in script's directory.

publish('script', 'options') provides a structure of options that may contain any of the following fields (first choice listed is the default):

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>'html'</td>
</tr>
<tr>
<td>stylesheet</td>
<td>''</td>
</tr>
<tr>
<td>outputDir</td>
<td>'' (an html subfolder below the file)</td>
</tr>
<tr>
<td>imageFormat</td>
<td>'png'</td>
</tr>
<tr>
<td>figureSnapMethod</td>
<td>'print'</td>
</tr>
</tbody>
</table>
Examples

To publish the file 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 a browser to view the published document.

See Also

    notebook

    Publishing to HTML, XML, LaTeX, Word and PowerPoint Using Cells
pwd

**Purpose**
Display 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`.

**See Also**
`cd`, `dir`, `fileparts`, `mfilename`, `path`, `what`
Purpose
Quasi-Minimal Residual method

Syntax

\[
\begin{align*}
x &= \text{qmr}(A,b) \\
\text{qmr}(A,b,tol) \\
\text{qmr}(A,b,tol,maxit) \\
\text{qmr}(A,b,tol,maxit,M) \\
\text{qmr}(A,b,tol,maxit,M1,M2) \\
\text{qmr}(A,b,tol,maxit,M1,M2,x0) \\
\text{qmr}(afun,b,tol,maxit,mfun,m2fun,x0,p1,p2,...) \\
[x,\text{flag}] &= \text{qmr}(A,b,...) \\
[x,\text{flag},\text{relres}] &= \text{qmr}(A,b,...) \\
[x,\text{flag},\text{relres},\text{iter}] &= \text{qmr}(A,b,...) \\
[x,\text{flag},\text{relres},\text{iter},\text{resvec}] &= \text{qmr}(A,b,...)
\end{align*}
\]

Description

\( x = \text{qmr}(A,b) \) attempts to solve the system of linear equations \( A\cdot x = b \) for \( x \).

The \( n \times n \) 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 \( afun \) such that \( afun(x) \) returns \( A\cdot x \) and \( afun(x,'\text{transp}') \) returns \( A'\cdot x \).

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

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

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

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

\( \text{qmr}(A,b,tol,maxit,M1,M2,x0) \) specifies the initial guess. If \( x0 \) is [], then \( \text{qmr} \) uses the default, an all zero vector.
qmr(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...) passes parameters p1,p2,... to functions afun(x,p1,p2,...) and afun(x,p1,p2,...,'transp') and similarly to the preconditioner functions m1fun and m2fun.

[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,[]);

Alternatively, use this matrix-vector product function

function y = afun(x,n,transp_flag)
if (nargin > 2) & strcmp(transp_flag,'transp')
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);
else
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

as input to qmr

x1 = qmr(@afun,b,tol,maxit,M1,M2,[],n);

Example 2.

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 $1 \times 10^{-6}$.

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), \ (backslash)

**References**

method for non-Hermitian linear systems”, *SIAM Journal: Numer. Math.* 60,
Purpose

Orthogonal-triangular decomposition

Syntax

\[ [Q,R] = \text{qr}(A) \] (full and sparse matrices)
\[ [Q,R] = \text{qr}(A,0) \] (full and sparse matrices)
\[ [Q,R,E] = \text{qr}(A) \] (full matrices)
\[ [Q,R,E] = \text{qr}(A,0) \] (full matrices)
\[ X = \text{qr}(A) \] (full matrices)
\[ R = \text{qr}(A) \] (sparse matrices)
\[ [C,R] = \text{qr}(A,B) \] (sparse matrices)
\[ R = \text{qr}(A,0) \] (sparse matrices)
\[ [C,R] = \text{qr}(A,B,0) \] (sparse matrices)

Description

The \textit{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 orthonormal or 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 \( A = Q \cdot R \). 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 \( A \cdot E = Q \cdot R \). The column permutation \( E \) is chosen so that \( \text{abs} (\text{diag}(R)) \) 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 \( A(:,E) = Q \cdot R \). The column permutation \( E \) is chosen so that \( \text{abs} (\text{diag}(R)) \) is decreasing.

\( X = \text{qr}(A) \) for full matrix \( A \), returns the output of the LAPACK subroutine \text{DGEQRF} or \text{ZGEQRF}. \text{triu}(\text{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:

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

$$Q = \begin{bmatrix} \end{bmatrix}$$
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

$$x =$$

0.5000
0
0.1667

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

$$\text{tol} = \max(\text{size}(A)) \times \text{eps} \times \text{abs}(R(1,1))$$

The solution $x$ was computed using the factorization and the two steps

$$y = Q' \ast b;$$
$$x = R \backslash y$$
The computed solution can be checked by forming $Ax$. This equals $b$ to within roundoff error, which indicates that even though the simultaneous equations $Ax = b$ 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**

The `qr` function uses LAPACK routines 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></td>
<td>DGEQRF</td>
<td>ZGEQRF</td>
</tr>
<tr>
<td><code>R = qr(A,0)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Real</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[Q,R] = qr(A)</code></td>
<td>DGEQRF, DORGQR</td>
<td>ZGEQRF, ZUNGQR</td>
</tr>
<tr>
<td><code>[Q,R] = qr(A,0)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Real</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[Q,R,e] = qr(A)</code></td>
<td>DGEQP3, DORGQR</td>
<td>ZGEQPF, ZUNGQR</td>
</tr>
<tr>
<td><code>[Q,R,e] = qr(A,0)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

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

The arithmetic operators \ and /

**References**

Purpose
Delete column or row from QR factorization

Syntax

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

Description

\[ [Q1,R1] = qrdelete(Q,R,j) \] 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 \( A \).

\[ [Q1,R1] = qrdelete(Q,R,j,'col') \] is the same as \( qrdelete(Q,R,j) \).

\[ [Q1,R1] = qrdelete(Q,R,j,'row') \] 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

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

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

\[
R1 = \\
0    & -19.7045 & -10.9891 & 0.4318  & -1.4873 \\
0    & 0  & 22.7444 & 5.8357  & -3.1977 \\
0    & 0  & 0   & -14.5784 & 3.7796 \\
\]

returns a valid QR factorization, although possibly different from

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

#### Q2

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.5274</td>
<td>0.5197</td>
<td>0.6697</td>
<td>0.0578</td>
<td></td>
</tr>
<tr>
<td>-0.7135</td>
<td>-0.6911</td>
<td>-0.0158</td>
<td>0.1142</td>
<td></td>
</tr>
<tr>
<td>-0.3102</td>
<td>0.1982</td>
<td>-0.4675</td>
<td>-0.8037</td>
<td></td>
</tr>
<tr>
<td>-0.3413</td>
<td>0.4616</td>
<td>-0.5768</td>
<td>0.5811</td>
<td></td>
</tr>
</tbody>
</table>

#### R2

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19.7045</td>
<td>10.9891</td>
<td>-0.4318</td>
<td>1.4873</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-22.7444</td>
<td>-5.8357</td>
<td>3.1977</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-14.5784</td>
<td>3.7796</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Algorithm

The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

#### See Also

`planerot`, `qr`, `qrinsert`
Insert column or row into QR factorization

\[
\begin{align*}
[Q_1, R_1] &= \text{qrinsert}(Q, R, j, x) \\
[Q_1, R_1] &= \text{qrinsert}(Q, R, j, x, 'col') \\
[Q_1, R_1] &= \text{qrinsert}(Q, R, j, x, 'row')
\end{align*}
\]

The function qrinsert returns the QR factorization of the matrix \( A_1 \), where \( A_1 \) is \( A = QR \) with the column \( x \) inserted before \( A(:, j) \). If \( A \) has \( n \) columns and \( j = n+1 \), then \( x \) is inserted after the last column of \( A \).

\[
[Q_1, R_1] = \text{qrinsert}(Q, R, j, x, 'col') \text{ is the same as } [Q_1, R_1] = \text{qrinsert}(Q, R, j, x).
\]

\[
[Q_1, R_1] = \text{qrinsert}(Q, R, j, x, 'row') \text{ returns the QR factorization of the matrix } A_1, \text{ where } A_1 \text{ is } A = QR \text{ with an extra row, } x, \text{ inserted before } A(j, :).
\]

**Examples**

\[
A = \text{magic}(5); \\
[Q, R] = \text{qr}(A); \\
\text{j} = 3; \\
x = 1:5; \\
[Q_1, R_1] = \text{qrinsert}(Q, R, j, x, 'row')
\]

\[
Q_1 = \\
\begin{bmatrix}
0.5231 & 0.5039 & -0.6750 & 0.1205 & 0.0411 & 0.0225 \\
0.7078 & -0.6966 & 0.0190 & -0.0788 & 0.0833 & -0.0150 \\
0.0308 & 0.0592 & 0.0656 & 0.1169 & 0.1527 & -0.9769 \\
0.1231 & 0.1363 & 0.3542 & 0.6222 & 0.6398 & 0.2104 \\
0.3077 & 0.1902 & 0.4100 & 0.4161 & -0.7264 & -0.0150 \\
0.3385 & 0.4500 & 0.4961 & -0.6366 & 0.1761 & 0.0225
\end{bmatrix}
\]

\[
R_1 = \\
\begin{bmatrix}
0 & 19.9292 & 12.4403 & 2.1340 & 4.3271 \\
0 & 0 & 24.4514 & 11.8132 & 3.9931 \\
0 & 0 & 0 & 20.2382 & 10.3392 \\
0 & 0 & 0 & 0 & 16.1948 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

returns a valid QR factorization, although possibly different from
A2 = [A(1:j-1,:); x; A(j:end,:)];
[Q2,R2] = qr(A2)

Q2 =
[-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]

R2 =
[  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      0]

**Algorithm**
The `qrinsert` function inserts the values of x into the jth column (row) of R. It then uses a series of Givens rotations to zero out the nonzero elements of R on and below the diagonal in the jth column (row).

**See Also**
`planerot`, `qr`, `qrdelete`
**Description**

Rank 1 update to QR factorization

**Syntax**

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

**Description**

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

\text{qrupdate} works only for full matrices.

**Examples**

The matrix

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

\[
\mu = \\
1.4901\times 10^{-08}
\]

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

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 \(R\).

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

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 \(\text{sqrt}(\text{eps})\).

Consider the update vectors

\[
u = [-1\ 0\ 0\ 0]'\; ; \; v = \text{ones}(4,1);
\]
Instead of computing the rather trivial QR factorization of this rank one update to \( A \) from scratch with

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

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

\[
RT = 
\begin{bmatrix}
1.0e-007 & \ast \\
-0.1490 & 0 & 0 & 0 \\
0 & -0.1490 & 0 & 0 \\
0 & 0 & -0.1490 & 0 \\
0 & 0 & 0 & -0.1490 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

we may use \texttt{qrupdate}.

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

\[
Q1 =
\begin{bmatrix}
-0.0000 & -0.0000 & -0.0000 & -0.0000 & 1.0000 \\
1.0000 & -0.0000 & -0.0000 & -0.0000 & 0.0000 \\
0.0000 & 1.0000 & -0.0000 & -0.0000 & 0.0000 \\
0.0000 & 0.0000 & 1.0000 & -0.0000 & 0.0000 \\
-0.0000 & -0.0000 & -0.0000 & 1.0000 & 0.0000 \\
\end{bmatrix}
\]

\[
R1 = 
\begin{bmatrix}
1.0e-007 & \ast \\
0.1490 & 0.0000 & 0.0000 & 0.0000 \\
0 & 0.1490 & 0.0000 & 0.0000 \\
0 & 0 & 0.1490 & 0.0000 \\
\end{bmatrix}
\]
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

Note

The quad8 function, which implemented a higher order method, is obsolete. The quadl function is its recommended replacement.

Syntax

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

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 for either an M-file function or an anonymous function. The function \( y = f(x) \) 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 online MATLAB documentation, explains how to provide addition 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.

**Examples**

Pass M-file function handle `@myfun` to `quad`:

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

where the M-file `myfun.m` is

```matlab
function y = myfun(x)
    y = 1./(x.^3-2*x-5);
end
```

Pass anonymous function handle `F` to `quad`:

```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, quad2, quad3, @ (function handle), anonymous functions

**References**

**Purpose**

Numerically evaluate integral, adaptive Lobatto quadrature

**Syntax**

$q = \text{quadl}(\text{fun},a,b)$
$q = \text{quadl}(\text{fun},a,b,\text{tol})$
$q = \text{quadl}(\text{fun},a,b,\text{tol},\text{trace})$
$[q,\text{fcnt}] = \text{quadl}(\text{fun},a,b,...)$

**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 for either an M-file function or an anonymous function. $\text{fun}$ accepts a vector $x$ and returns a vector $y$, the function $\text{fun}$ evaluated at each element of $x$.

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

$q = \text{quadl}(\text{fun},a,b,\text{tol})$ 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.

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

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

Use array operators .* , . / and .^ 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 quad1:

$Q = \text{quadl}(@\text{myfun},0,2);$  
where the M-file myfun.m is

function $y = \text{myfun}(x)$
$y = 1./(x.^3-2*x-5);$  

Pass anonymous function handle $\text{F}$ to quad1:
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 = quadv(fun,A,B)
Q = quadv(fun,A,B,tol)
Q = quadv(fun,A,B,tol,trace)
[Q,fcnt] = quadv(...)  

Description
Q = quadv(fun,A,B) approximates the integral of the complex array-valued function fun from A to B to within an error of 1.e-6 using recursive adaptive Simpson quadrature. The function y = fun(x) should accept a scalar argument x and return an array result Y, whose components are the integrands evaluated at x.

Parameterizing Functions Called by Function Functions, in the online MATLAB documentation, explains how to provide addition parameters to the function fun, if necessary.

Q = quadv(fun,A,B,tol) uses the absolute error tolerance TOL for all the integrals instead of the default, which is 1.e-6.

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

[Q,fcnt] = quadv(...) returns the number of function evaluations.

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

Example

fun = @(x,n) (1./((1:n)+x));
Q = quadv(fun,0,1,[],[],10)

The resulting array Q has elements Q(k) = log((k+1)./(k)).

Q =

Columns 1 through 8

0.6931          0.4055          0.2877          0.2231          0.1823          0.1542
0.1335          0.1178

Columns 9 through 10
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1054</td>
<td>0.0953</td>
</tr>
</tbody>
</table>

**See Also**
quad, dblquad, triplequad
Purpose
Create and display question dialog box

Syntax
button = questdlg('qstring')
button = questdlg('qstring','title')
button = questdlg('qstring','title','default')
button = questdlg('qstring','title','str1','str2','default')
button = questdlg('qstring','title','str1','str2','str3','default')

Description
button = questdlg('qstring') displays a modal dialog 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.

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'.

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 Return key displays a warning and the dialog remains open.

See Also
inputdlg, textwrap
Purpose

Terminate MATLAB

Graphical Interface

As an alternative to the quit function, use the close box or select Exit MATLAB from the File menu in the MATLAB desktop.

Syntax

quit
quit cancel
quit force

Description

quit 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 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 uiswait, waitfor, or drawnow so that figures are visible. See the reference pages for these functions for more information.
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

`finish`, `save`, `startup`

Purpose

Quiver or velocity plot

Syntax

- `quiver(x,y,u,v)`
- `quiver(u,v)`
- `quiver(...,scale)`
- `quiver(...,LineSpec)`
- `quiver(...,LineSpec,'filled')`
- `quiver(axes_handle,...)`
- `h = quiver(...)`
- `hlines = quiver('v6',...)`

Description

A quiver plot displays velocity vectors as arrows with components \((u,v)\) at the points \((x,y)\).

For example, the first vector is defined by components \(u(1),v(1)\) and is displayed at the point \(x(1),y(1)\).

`quiver(x,y,u,v)` plots vectors as arrows at the coordinates specified in each corresponding pair of elements in \(x\) and \(y\). The matrices \(x, y, u,\) and \(v\) must all
be the same size and contain corresponding position and velocity components. However, \( x \) and \( y \) can also be vectors, as explained in the next section.

**Expanding \( x \) and \( y \) Coordinates**
MATLAB expands \( x \) and \( y \) if they are not matrices. This expansion is equivalent to calling `meshgrid` to generate matrices from vectors:

\[
[x,y] = \text{meshgrid}(x,y);
\text{quiver}(x,y,u,v)
\]

In this case, the following must be true:

\[
\text{length}(x) = n \quad \text{and} \quad \text{length}(y) = m, \quad \text{where} \quad [m,n] = \text{size}(u) = \text{size}(v).
\]

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

\text{quiver}(...,\text{scale}) \quad \text{automatically scales the arrows to fit within the grid and then stretches them by the factor} \quad \text{scale}. \quad \text{scale} = 2 \quad \text{doubles their relative length and} \quad \text{scale} = 0.5 \quad \text{halves the length. Use} \quad \text{scale} = 0 \quad \text{to plot the velocity vectors without automatic scaling.}

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

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

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

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

**Backward Compatible Version**

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

Showing the Gradient with Quiver Plots

Plot the gradient field of the function \( z = xe^{-(x^2+y^2)} \):

\[
\begin{align*}
[X,Y] &= \text{meshgrid}(-2:.2:2); \\
Z &= X.*\exp(-X.^2 - Y.^2); \\
[DX,DY] &= \text{gradient}(Z,.2,.2); \\
\text{contour}(X,Y,Z) \\
\text{hold on} \\
\text{quiver}(X,Y,DX,DY) \\
\text{colormap hsv} \\
\text{hold off}
\end{align*}
\]

See Also

contour, LineSpec, plot, quiver3

“Direction and Velocity Plots” for related functions

Two-Dimensional Quiver Plots for more examples

See “Quivergroup Properties” for property descriptions
Purpose
Three-dimensional velocity plot

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, 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`.

- `quiver3(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

- `quiver3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

- `h = quiver3(...)` returns a vector of line handles.

Examples
Plot the surface normals of the function $z = xe^{(-x^2-y^2)}$.

```matlab
[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
```
quiver3

\[ [U,V,W] = \text{surfnorm}(X,Y,Z); \]
\[ \text{quiver3}(X,Y,Z,U,V,W,0.5); \]
\[ \text{hold on} \]
\[ \text{surf}(X,Y,Z); \]
\[ \text{colormap hsv} \]
\[ \text{view}(-35,45) \]
\[ \text{axis}([-2 2 -1 1 -.6 .6]) \]
\[ \text{hold off} \]

See Also

axis, contour, LineSpec, plot, plot3, quiver, surfnorm, view

“Direction and Velocity Plots” for related functions

Three-Dimensional Quiver Plots for more examples
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 area series objects.

See Plot Objects for more information on quiver group objects.

Quivergroup Property Descriptions

This section provides a description of properties. Curly braces { } enclose default values.

AutoScale {on} | off

*Autoscale arrow length.* Based on average spacing in the x and y directions, AutoScale scales the arrow length to fit within the grid-defined coordinate data and keeps the arrows from overlapping. After autoscaling, quiver applies the AutoScaleFactor to the arrow length.

AutoScaleFactor scalar (default = 0.9)

*User-specified scale factor.* When AutoScale is on, the quiver function applies this user-specified autoscale factor to the arrow length. A value of 2 doubles the length of the arrows; 0.5 halves the length.

BeingDeleted on | {off} Read Only

*This object is being deleted.* The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the BeingDeleted property to on when the object’s delete function callback is called (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 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 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
Quivergroup Properties

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 the quivergroup object.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

The expression executes in the MATLAB workspace.

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

Children array of graphics object handles

Children of the quivergroup object. An array containing the handles of all line objects parented to the quivergroup 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 the quiver Children property unless you set the Root ShowHiddenHandles property to on:

```
set(0,'ShowHiddenHandles','on')
```

Clipping {on} | off

Clipping mode. MATLAB clips quiver plots to the axes plot box by default. If you set Clipping to off, arrows might be displayed outside the axes plot box.

Color ColorSpec

Color of arrows. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color. See the ColorSpec reference page for more
Quivergroup Properties

information on specifying color. For example, the following statement shows
the arrow color set to blue.

\[ h = \text{quiver}(u,v,'Color','b'); \]

CreateFcn string or function handle

Callback routine executed during object creation. This property defines a
callback that executes when MATLAB creates a quivergroup object. You must
specify the callback during the creation of the object. For example,

\[ \text{quiver}(u,v,'CreateFcn',@CallbackFcn) \]

where @CallbackFcn is a function handle that references the callback function.
MATLAB executes this routine after setting all other quivergroup properties.
Setting this property on an existing quivergroup 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 the
quivergroup object is deleted (e.g., this might happen when you issue a delete
command on the quivergroup object, its parent axes, or the figure containing
it). MATLAB executes the callback before destroying the object’s properties so
that 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 and the plot browser use this text for
labels for any quivergroup objects appearing in these legends.
Erase mode. This property controls the technique MATLAB uses to draw and erase quiver child objects (the lines used to construct the arrows). 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 might mathematically combine layers of colors (e.g., performing an XOR operation on a pixel color and the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.
Quivergroup Properties

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 the quivergroup object.

- 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.
**Quivergroup Properties**

**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.

**HitTest**
{on} | off

*Selectable by mouse click.* `HitTest` determines whether the quivergroup 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 line objects that compose the quiver plot. If `HitTest` is `off`, clicking the quivergroup object selects the object below it (which is usually the axes containing it).

**HitTestArea**
on | {off}

*Select quivergroup object on arrows or extent of graph.* This property enables you to select quivergroup objects in two ways:

- Select by clicking quiver arrows (default).
- Select by clicking anywhere in the extent of the quiver plot.

When `HitTestArea` is `off`, you must click the quiver lines (excluding the base line) to select the quivergroup object. When `HitTestArea` is `on`, you can select the quivergroup object by clicking anywhere within the extent of the graph (i.e., anywhere within a rectangle that encloses all the arrows).

**Interruptible**
{on} | off

*Callback routine interruption mode.* The `Interruptible` property controls whether a quivergroup object callback can be interrupted by subsequently invoked callbacks.

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 quiver 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 used for the quiver arrows. Available line styles are shown in the following table.

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

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

*Width of the quiver arrows.* 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 the x- and y-coordinates. 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>
</tbody>
</table>
### Quivergroup 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 quiver 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).

**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.
Quivergroup Properties

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 axes handle

*Parent of quivergroup object.* This property contains the handle of the quivergroup object’s parent object. The parent of a quivergroup 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 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 the quivergroup object is selected.

SelectionHighlight {on} | off

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing selection handles on the arrows. When SelectionHighlight is off, MATLAB does not draw the handles.

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.

For example, you might create a quivergroup object and set the Tag property:

\[ t = \text{quiver}(u,v,'Tag','\text{'quiver1'}') \]
Quivergroup Properties

When you want to access the quivergroup object, you can use findobj to find the object’s handle. The following statement changes the Color property of the object whose Tag is quiver1.

```matlab
set(findobj('Tag','quiver1'),'Color','red')
```

**Type**

string (read only)

*Type of graphics object.* This property contains a string that identifies the class of the graphics object. For stem objects, Type is 'hggroup'. This statement finds all the hggroup objects in the current axes.

```matlab
t = findobj(gca,'Type','hggroup');
```

**UIContextMenu**

handle of a uicontextmenu object

*Associate a context menu with the quivergroup object.* Assign this property the handle of a uicontextmenu object created in the quivergroup object's parent figure. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click over the extent of the quivergroup object.

**UserData**

array

*User-specified data.* This property can be any data you want to associate with the quivergroup object (including cell arrays and structures). The quivergroup object does not set values for this property, but you can access it using the set and get functions.

**Visible**

{on} | off

*Visibility of quivergroup object and its children.* By default, stem object visibility is on. This means all children of the quivergroup object are visible unless the child object’s Visible property is set to off. Setting a quivergroup object’s Visible property to off also makes its children invisible.

**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).
Quivergroup Properties

**UDataSource**

string (MATLAB variable)

*Link UData to MATLAB variable.* Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the UData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change UData.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

**Note**  If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

**VData**

matrix

*One dimension of 2-D or 3-D vector components.* UData, VData and WData (for 3-D) together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components UData(1), VData(1), WData(1).

**VDataSource**

string (MATLAB variable)

*Link VData to MATLAB variable.* Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the VData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change VData.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.
Quivergroup Properties

**Note**  If you change one data source property to return data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

**WData**  
**matrix**

_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**

_2-D or 3-D vector coordinates._ 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, \( \text{length(XData)} = \text{size(UData,2)} \).
Quivergroup Properties

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.

**XDataSource**

string (MATLAB variable)

Link XData to MATLAB variable. Set this property to a MATLAB variable that, by default, 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 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.

**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 \( Y \)Data (i.e., the input argument \( Y \)), the quiver function uses the indices of \( V \)Data to create the quiver graph. See the \( Y \)DataMode property for related information.

The input argument \( y \) in the quiver function calling syntax assigns values to \( Y \)Data.

\[ Y \text{DataMode} \quad \{ \text{auto} \mid \text{manual} \} \]

Use automatic or user-specified \( y \)-axis values. If you specify \( Y \)Data (by setting the \( Y \)Data property or specifying the input argument \( Y \)), MATLAB sets this property to manual.

If you set \( Y \)DataMode to auto after having specified \( Y \)Data, MATLAB resets the \( y \) tick-mark labels to the indices of the \( U \), \( V \), and \( W \) data, overwriting any previous values.

\[ Y \text{DataSource} \quad \text{string (MATLAB variable)} \]

Link \( Y \)Data to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the \( Y \)Data.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change \( Y \)Data.

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.

\[ Z \text{Data} \quad \text{vector or matrix} \]

\( Z \)-axis coordinates of arrows. The quiver function draws an individual arrow at each \( z \)-axis location in the \( Z \)Data array. \( Z \)Data must be a matrix equal in size to \( X \)Data and \( Y \)Data.
The input argument \( z \) in the \texttt{quiver3} function calling syntax assigns values to \( Z\text{Data} \).
Purpose
QZ factorization for generalized eigenvalues

Syntax

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

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

\[ [\text{AA}, \text{BB}, \text{Q}, \text{Z}] = \text{qz}(A, B) \]
for square matrices \( A \) and \( B \), produces upper quasitriangular matrices \( \text{AA} \) and \( \text{BB} \), and unitary matrices \( \text{Q} \) and \( \text{Z} \) such that \( \text{Q}^*A*\text{Z} = \text{AA} \), and \( \text{Q}^*B*\text{Z} = \text{BB} \). For complex matrices, \( \text{AA} \) and \( \text{BB} \) are triangular.

\[ [\text{AA}, \text{BB}, \text{Q}, \text{Z}, \text{V}, \text{W}] = \text{qz}(A, B) \]
also produces matrices \( \text{V} \) and \( \text{W} \) 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 \( \text{AA} \). For compatibility with earlier versions, `'complex'` is the default.
- `'real'`: Produces a real decomposition with a quasitriangular \( \text{AA} \), containing 1-by-1 and 2-by-2 blocks on its diagonal.

If \( \text{AA} \) is triangular, the diagonal elements of \( \text{AA} \) and \( \text{BB} \), \( \alpha = \text{diag}(\text{AA}) \) and \( \beta = \text{diag}(\text{BB}) \), 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 \( \text{AA} \) is triangular, the diagonal elements of \( \text{AA} \) and \( \text{BB} \),
Purpose
Uniformly distributed random numbers and arrays

Syntax
\[
Y = \text{rand}(n) \\
Y = \text{rand}(m,n) \\
Y = \text{rand}([m \ n]) \\
Y = \text{rand}(m,n,p,...) \\
Y = \text{rand}([m \ n \ p...]) \\
Y = \text{rand(size(A))} \\
\text{rand} \\
s = \text{rand('state')} \\
\]

Description
The \text{rand} function generates arrays of random numbers whose elements are uniformly distributed in the interval \((0,1)\).

\(Y = \text{rand}(n)\) returns an \(n\)-by-\(n\) matrix of random entries. An error message appears if \(n\) is not a scalar.

\(Y = \text{rand}(m,n)\) or \(Y = \text{rand}([m \ n])\) returns an \(m\)-by-\(n\) matrix of random entries.

\(Y = \text{rand}(m,n,p,...)\) or \(Y = \text{rand}([m \ n \ p...])\) generates random arrays.

\(Y = \text{rand(size(A))}\) returns an array of random entries that is the same size as \(A\).

\text{rand}, by itself, returns a scalar whose value changes each time it’s referenced.

\(s = \text{rand('state')}\) returns a 35-element vector containing the current state of the uniform generator. To change the state of the generator:

\[
\text{rand('state',s)} \quad \text{Resets the state to } s. \\
\text{rand('state',0)} \quad \text{Resets the generator to its initial state.} \\
\text{rand('state',j)} \quad \text{For integer } j, \text{ resets the generator to its } j\text{-th state.} \\
\text{rand('state',sum(100*clock))} \quad \text{Resets it to a different state each time.} \\
\]
**Examples**

**Example 1.** \( R = \text{rand}(3,4) \) may produce

\[
\begin{bmatrix}
0.2190 & 0.6793 & 0.5194 & 0.0535 \\
0.0470 & 0.9347 & 0.8310 & 0.5297 \\
0.6789 & 0.3835 & 0.0346 & 0.6711
\end{bmatrix}
\]

This code makes a random choice between two equally probable alternatives.

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

**Example 2.** 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 =
```

\[
\begin{bmatrix}
18.1106 & 10.6110 & 26.7460 & 43.5247 & 30.1125 \\
17.9489 & 39.8714 & 43.8489 & 10.7856 & 38.3789 \\
34.1517 & 27.8039 & 31.0061 & 37.2511 & 27.1557 \\
20.8875 & 47.2726 & 18.1059 & 25.1792 & 22.1847 \\
17.9526 & 28.6398 & 36.8855 & 43.2718 & 17.5861
\end{bmatrix}
\]

### See Also

randn, randperm, sprand, sprandn
Purpose
Normally distributed random numbers and arrays

Syntax

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

Description
The \text{randn} function generates arrays of random numbers whose elements are normally distributed with mean 0, variance $\sigma^2 = 1$, and standard deviation $\sigma = 1$.

$Y = \text{randn}(n)$ returns an $n$-by-$n$ matrix of random entries. An error message appears if $n$ is not a scalar.

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

$Y = \text{randn}(m,n,[p,...])$ or $Y = \text{randn}([m \ n \ p,...])$ generates random arrays.

$Y = \text{randn(size(A))}$ returns an array of random entries that is the same size as $A$.

\text{randn}, by itself, returns a scalar whose value changes each time it's referenced.

$s = \text{randn('state')}$ returns a 2-element vector containing the current state of the normal generator. To change the state of the generator:

\[
\text{randn('state',s)} \\
\text{randn('state',0)} \\
\text{randn('state',j)} \\
\text{randn('state',sum(100*clock))} \\
\]

Resets the state to $s$.
Resets the generator to its initial state.
For integer $j$, resets the generator to its $j$th state.
Resets it to a different state each time.
**Examples**

**Example 1.** \( R = \text{randn}(3,4) \) may produce

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

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

**Example 2.** Generate a random distribution with a specific mean and variance \( \sigma^2 \). To do this, multiply the output of \( \text{randn} \) 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

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

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

**See Also** \texttt{rand}, \texttt{randperm}, \texttt{sprand}, \texttt{sprandn}
**Purpose**
Random permutation

**Syntax**
p = randperm(n)

**Description**
p = randperm(n) returns a random permutation of the integers 1:n.

**Remarks**
The `randperm` function calls `rand` and therefore changes `rand`'s state.

**Examples**
randperm(6) might be the vector

```
[3 2 6 4 1 5]
```

or it might be some other permutation of 1:6.

**See Also**
`permute`
**Purpose**

Rank of a matrix

**Syntax**

\[
\begin{align*}
k &= \text{rank}(A) \\
k &= \text{rank}(A, \text{tol})
\end{align*}
\]

**Description**

The `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, \(\max(\text{size}(A)) \times \text{norm}(A) \times \text{eps}\).

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

returns the number of singular values of \(A\) that are larger than \(\text{tol}\).

**Remark**

Use `sprank` to determine the structural rank of a sparse matrix.

**Algorithm**

There are a number of ways to compute the rank of a matrix. MATLAB 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

\[
\begin{align*}
s &= \text{svd}(A) \\
tol &= \max(\text{size}(A)) \times s(1) \times \text{eps} \\
r &= \text{sum}(s > \text{tol})
\end{align*}
\]

**See Also**

`sprank`

**References**

**Purpose**
Rational fraction approximation

**Syntax**

```matlab
[N,D] = rat(X)
[N,D] = rat(X,tol)
rat(...)  
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

```matlab
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7
```

produces

```matlab
s = 0.7595
```
However, with

\texttt{format rat}

or with

\texttt{rats(s)}

the printed result is

\[ s = \frac{319}{420} \]

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity \( s \) is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement \texttt{rat(s)} produces

\[ 1 + \frac{1}{(-4 + 1/(-6 + 1/(-3 + 1/(-5))))} \]

And the statement

\[ [n,d] = \texttt{rat(s)} \]

produces

\[ n = 319, \ d = 420 \]

The mathematical quantity \( \pi \) is certainly not a rational number, but the MATLAB quantity \texttt{pi} that approximates it is a rational number. \texttt{pi} is the ratio of a large integer and \( 2^{52} \):

\[ \frac{14148475504056880}{4503599627370496} \]

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

\[ \frac{355}{113} \]

This approximation was known in Euclid’s time. Its decimal representation is

\[ 3.14159292035398 \]
and so it agrees with \( \pi \) to seven significant figures. The statement

\[ \text{rat}(\pi) \]

produces

\[ 3 + \frac{1}{(7 + \frac{1}{16})} \]

This shows how the \( \frac{355}{113} \) was obtained. The less accurate, but more familiar approximation \( \frac{22}{7} \) 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

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

The \( d \)s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when \( X = \sqrt{2} \). For \( x = \sqrt{2} \), the error with \( k \) terms is about \( 2.68 \times (.173)^k \), so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

**See Also**

format
rbbox

**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:
rbbox

- 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” 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: rcond uses LAPACK routines to compute the estimate of the reciprocal condition number:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>DLANGE, DGETRF, DGECON</td>
</tr>
<tr>
<td>Complex</td>
<td>ZLANGE, ZGETRF, ZGECO</td>
</tr>
</tbody>
</table>

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

References:
Purpose
Real part of complex number

Syntax
X = real(Z)

Description
X = real(Z) returns the real part of the elements of the complex array Z.

Examples
real(2+3i) is 2.

See Also
abs, angle, conj, i, j, imag
**Purpose**

Natural logarithm for nonnegative real arrays

**Syntax**

\[ Y = \text{reallog}(X) \]

**Description**

\[ Y = \text{reallog}(X) \]

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**

\[ M = \text{magic}(4) \]

\[
M =
\begin{bmatrix}
 16 & 2 & 3 & 13 \\
 5 & 11 & 10 & 8 \\
 9 & 7 & 6 & 12 \\
 4 & 14 & 15 & 1 \\
\end{bmatrix}
\]

\[ \text{reallog}(M) \]

\[
\text{ans} =
\begin{bmatrix}
 2.7726 & 0.6931 & 1.0986 & 2.5649 \\
 1.6094 & 2.3979 & 2.3026 & 2.0794 \\
 2.1972 & 1.9459 & 1.7918 & 2.4849 \\
 1.3863 & 2.6391 & 2.7081 & 0 \\
\end{bmatrix}
\]

**See Also**

log, realpow, realsqrt
Purpose
Largest positive floating-point number

Syntax
\[ n = \text{realmax} \]

Description
\[ n = \text{realmax} \] returns the largest floating-point number representable on your computer. Anything larger overflows.

\[ \text{realmax('double')} \] is the same as \[ \text{realmax} \] with no arguments.

\[ \text{realmax('single')} \] is the largest single precision floating point number representable on your computer. Anything larger overflows to \( \text{single(Inf)} \).

Examples
\[ \text{realmax} \] is one bit less than \( 2^{1024} \) or about \( 1.7977\times10^{308} \).

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

Execute \text{type} \[ \text{realmax} \] to see \( \text{maxexp} \) for various computers.

See Also
\[ \text{eps}, \text{realmin}, \text{intmax} \]
Purpose  Smallest positive floating-point number
Syntax  \( n = \text{realmin} \)
Description  \( n = \text{realmin} \) returns the smallest positive normalized floating-point number on your computer. Anything smaller underflows or is an IEEE “denormal.”

\( \text{REALMIN('double')} \) is the same as \( \text{REALMIN} \) with no arguments.

\( \text{REALMIN('single')} \) is the smallest positive normalized single precision floating point number on your computer.

Examples  \( \text{realmin} \) is \( 2^{-1022} \) or about \( 2.2251 \times 10^{-308} \).

Algorithm  The \( \text{realmin} \) function is equivalent to \( \text{pow2}(1,\text{minexp}) \) where \( \text{minexp} \) is the smallest possible floating-point exponent.

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

See Also  \( \text{eps, realmax, intmin} \)
Purpose
Array power for real-only output

Syntax
\[ Z = \text{realpow}(X,Y) \]

Description
\( Z = \text{realpow}(X,Y) \) raises each element of array \( X \) to the power of its corresponding element in array \( Y \). Arrays \( X \) and \( Y \) must be the same size. The range of \text{realpow} \ is the set of all real numbers, i.e., all elements of the output array \( Z \) must be real.

Examples
\[
X = -2*\text{ones}(3,3)
\]

\[
X =
\begin{bmatrix}
-2 & -2 & -2 \\
-2 & -2 & -2 \\
-2 & -2 & -2 \\
\end{bmatrix}
\]

\[
Y = \text{pascal}(3)
\]

\[
\text{ans} =
\begin{bmatrix}
1 & 1 & 1 \\
1 & 2 & 3 \\
1 & 3 & 6 \\
\end{bmatrix}
\]

\[
\text{realpow}(X,Y)
\]

\[
\text{ans} =
\begin{bmatrix}
-2 & -2 & -2 \\
-2 & 4 & -8 \\
-2 & -8 & 64 \\
\end{bmatrix}
\]

See Also
\text{reallog}, \text{realsqrt}, .^\text{(array power operator)}
Purpose
Square root for nonnegative real arrays

Syntax
Y = realsqrt(X)

Description
Y = realsqrt(X) returns the square root of each element of array X. Array X must contain only nonnegative real numbers. The size of Y is the same as the size of X.

Examples
M = magic(4)

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

realsqrt(M)

ans =
  4.0000    1.4142    1.7321    3.6056
  2.2361    3.3166    3.1623    2.8284
  3.0000    2.6458    2.4495    3.4641
  2.0000    3.7417    3.8730    1.0000

See Also
reallog, realpow, sqrt, sqrtm
Purpose
Create a 2-D rectangle object

Syntax
rectangle
rectangle('Position', [x, y, w, h])
rectangle(..., 'Curvature', [x, y])
h = rectangle(...)

Description
rectangle draws a rectangle with Position [0, 0, 1, 1] and Curvature [0, 0] (i.e., no curvature).

rectangle('Position', [x, y, w, h]) draws the rectangle from the point x, y and having a width of w and a height of h. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the x and y axes. You can do this with the command axis equal or daspect([1,1,1]).

rectangle(..., 'Curvature', [x, y]) specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature x is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature y is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of x and y can range from 0 (no curvature) to 1 (maximum curvature). A value of [0, 0] creates a rectangle with square sides. A value of [1,1] creates an ellipse. If you specify only one value for Curvature, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

h = rectangle(...) returns the handle of the rectangle object created.

Remarks
Rectangle objects are 2-D and can be drawn in an axes only if the view is [0 90] (i.e., view(2)). Rectangles are children of axes and are defined in coordinates of the axes data.

Examples
This example sets the data aspect ratio to [1,1,1] so that the rectangle is displayed in the specified proportions (daspect). Note that the horizontal and
vertical curvature can be different. Also, note the effects of using a single value for Curvature.

```matlab
rectangle('Position',[0.59,0.35,3.75,1.37],...
    'Curvature',[0.8,0.4],...
    'LineWidth',2,'LineStyle','--')
daspect([1,1,1])
```

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.

```plaintext
rectangle('Position',[1,2,5,10],'Curvature',[1,1],...
    'FaceColor','r')
daspect([1,1,1])
xlim([0,7])
ylim([1,13])
```
See Also

- line, patch, rectangle properties
- “Object Creation Functions” for related functions

See the annotation function for information about the rectangle annotation object.

Object Hierarchy
Setting Default Properties
You can set default rectangle properties on the axes, figure, and root levels:

```
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.

Property List
The following table lists all rectangle properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curvature</td>
<td>Degree of horizontal and vertical curvature</td>
<td>Value: two-element vector with values between 0 and 1 Default: [0,0]</td>
</tr>
<tr>
<td>EraseMode</td>
<td>Method of drawing and erasing the rectangle (useful for animation)</td>
<td>Values: normal, none, xor, background Default: normal</td>
</tr>
<tr>
<td>EdgeColor</td>
<td>Color of rectangle edges</td>
<td>Value: ColorSpec or none Default: ColorSpec [0,0,0]</td>
</tr>
<tr>
<td>FaceColor</td>
<td>Color of rectangle interior</td>
<td>Value: ColorSpec or none Default: none</td>
</tr>
<tr>
<td>LineStyle</td>
<td>Line style of edges</td>
<td>Values: -, --, :, --, none Default: –</td>
</tr>
<tr>
<td>LineWidth</td>
<td>Width of edge lines in points</td>
<td>Value: scalar Default: 0.5 points</td>
</tr>
<tr>
<td>Position</td>
<td>Location and width and height of rectangle</td>
<td>Value: [x,y,width,height] Default: [0,0,1,1]</td>
</tr>
</tbody>
</table>
### General Information About Rectangle Objects

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Children</td>
<td>Rectangle objects have no children.</td>
<td></td>
</tr>
<tr>
<td>Parent</td>
<td>The parent of a rectangle object is an axes, hggroup, or hgtransform object.</td>
<td>Value: object handle</td>
</tr>
</tbody>
</table>
| Selected      | Indicates if the rectangle is in a selected state | Values: on, off  
Default: off |
| Tag           | User-specified label | Value: any string  
Default: '' (empty string) |
| Type          | The type of graphics object (read only) | Value: the string 'rectangle' |
| UserData      | User-specified data | Value: any matrix  
Default: [] (empty matrix) |

### Properties Related to Callback Routine Execution

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| BeingDeleted       | Query to see if object is being deleted.                                             | Values: on | off  
Read only |
| BusyAction         | Specifies how to handle callback routine interruption                                | Values: cancel, queue  
Default: queue |
| ButtonDownFcn      | Defines a callback routine that executes when a mouse button is pressed on over the rectangle | Value: string or function handle  
Default: '' (empty string) |
| CreateFcn          | Defines a callback routine that executes when a rectangle is created                  | Value: string or function handle  
Default: '' (empty string) |
| DeleteFcn          | Defines a callback routine that executes when the rectangle is deleted (via close or delete) | Value: string or function handle  
Default: '' (empty string) |
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interruptible</td>
<td>Determines if callback routine can be interrupted</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on (can be interrupted)</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Associates a context menu with the rectangle</td>
<td>Value: handle of a Uicontextmenu</td>
</tr>
</tbody>
</table>

**Controlling Access to Objects**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandleVisibility</td>
<td>Determines if and when the rectangle's handle is visible to other functions</td>
<td>Values: on, callback, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>HitTest</td>
<td>Determines if the rectangle can become the current object (see the Figure CurrentObject property)</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
</tbody>
</table>

**Controlling the Appearance**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clipping</td>
<td>Clipping to axes rectangle</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Highlights rectangle when selected (Selected property is set to on)</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>Visible</td>
<td>Makes the rectangle visible or invisible</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
</tbody>
</table>
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 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**  
string or function handle

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

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**  
string or function handle

*Callback routine executed during object creation.* This property defines a callback routine 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',
    'set(gca,''DataAspectRatio'',[1,1,1]))
```

defines a default value on the root level that sets the axes DataAspectRatio whenever you create a rectangle object. MATLAB executes this routine after setting all rectangle properties. Setting this property on an existing rectangle 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.

Curvature

one- or two-element vector [x; y]

Amount of horizontal and vertical curvature. This property specifies the curvature of the rectangle sides, which enables the shape of the rectangle to vary from rectangular to ellipsoidal. The horizontal curvature x is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature y is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of x and y can range from 0 (no curvature) to 1 (maximum curvature). A value of [0, 0] creates a rectangle with square sides. A value of [1, 1] creates an ellipse. If you specify only one value for Curvature, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

DeleteFcn

string or function handle

Delete rectangle callback routine. A callback routine that executes when you delete the rectangle object (e.g., when you issue a delete command or clear the axes or figure). 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.

EdgeColor

{ColorSpec} | none

Color of the rectangle edges. This property specifies the color of the rectangle edges as a color or specifies that no edges be drawn.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase rectangle objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- normal (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are
rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- **none** — Do not erase the rectangle when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode` none, you cannot print it because MATLAB stores no information about its former location.

- **xor** — Draw and erase the rectangle by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the rectangle. However, the rectangle's color depends on the color of whatever is beneath it on the display.

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

**Printing with Nonnormal Erase Modes.**

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

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

**FaceColor**

`ColorSpec | {none}`

*Color of rectangle face.* This property specifies the color of the rectangle face, which is not colored by default.

**HandleVisibility**

`{on} | callback | off`

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

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

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

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

You can set the `Root ShowHiddenHandles` property to `on` to make all handles visible regardless of their handle visibility 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**

*The width of the rectangle edge line.* Specify this value in points (1 point = \(1/72\) inch). The default LineWidth is 0.5 points.

**Parent**

*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.
**Rectangle properties**

**Selected**  on | off

*Is object selected?* When this property is on MATLAB displays selection handles if the `SelectionHighlight` property is also on. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**SelectionHighlight** {on} | off

*Objects are highlighted when selected.* When the `Selected` property is on, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is off, MATLAB does not draw the handles.

**Tag**  string

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

**Type**  string (read only)

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

**UIContextMenu**  handle of a `uicontextmenu` object

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

**UserData**  matrix

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

**Visible**  {on} | off

*Rectangle visibility.* By default, all rectangles are visible. When set to off, the rectangle is not visible, but still exists, and you can get and set its properties.
Purpose

Rectangle intersection area.

Syntax

area = rectint(A,B)

Description

area = rectint(A,B) returns the area of intersection of the rectangles specified by position vectors A and B.

If A and B each specify one rectangle, the output area is a scalar.

A and B can also be matrices, where each row is a position vector. area is then a matrix giving the intersection of all rectangles specified by A with all the rectangles specified by B. That is, if A is n-by-4 and B is m-by-4, then area is an n-by-m matrix where area(i,j) is the intersection area of the rectangles specified by the ith row of A and the jth row of B.

Note  A position vector is a four-element vector [x,y,width,height], where the point defined by x and y specifies one corner of the rectangle, and width and height define the size in units along the x and y axes respectively.

See Also

polyarea
Recycle

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 folder (on UNIX). 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
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.

Examples
Start from a state where file recycling has been turned off. Check the current recycle state:

    recycle
    ans =
    off

Turn file recycling on. Delete a file and verify that it has been transferred to the recycle bin or temporary folder:

    recycle on;
    delete myfile.txt
See Also
delete, dir, ls, fileparts, mkdir, rmdir
reducepatch

**Purpose**
Reduce the number of patch faces

**Syntax**
```
reducepatch(p,r)
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)
```

**Description**
`reducepatch(p,r)` reduces the number of faces of the patch identified by handle `p`, while attempting to preserve the overall shape of the original object. 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.

```matlab
[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)
```
Before Reduction
After Reduction to 15% of Original Number of Faces

See Also

isosurface, isocaps, isonormals, smooth3, subvolume, reducevolume

"Volume Visualization" for related functions

Vector Field Displayed with Cone Plots for another example
reducevolume

**Purpose**
Reduce the number of elements in a volume data set

**Syntax**

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])
```

```
[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])
```

```
nv = reducevolume(...)  
```

**Description**

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])` reduces the number of elements in the volume by retaining every `Rx`\textsuperscript{th} element in the `x` direction, every `Ry`\textsuperscript{th} element in the `y` direction, and every `Rz`\textsuperscript{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] = reducevolume(V,[Rx,Ry,Rz])` 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 = reducevolume(...) returns only the reduced volume.
```

**Examples**

This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every fourth element in the `x` and `y` directions and every element in the `z` direction.
- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).
A 100-element grayscale colormap provides coloring for the end caps (colormap).

Adding a light to the right of the camera illuminates the object (camlight, lighting).

```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');
isnormals(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” for related functions
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Redraw current figure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>refresh</code></td>
</tr>
<tr>
<td></td>
<td><code>refresh(h)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>refresh</code> erases and redraws the current figure.</td>
</tr>
<tr>
<td></td>
<td><code>refresh(h)</code> redraws the figure identified by h.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>“Figure Windows” for related functions</td>
</tr>
</tbody>
</table>
refreshdata

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 objects_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.

z = peaks(5);
[c h] = contour(z,'ZDataSource','z');
drawnow
pause(3) % Wait 3 seconds and the graph will update
z = peaks(20);
refreshdata(h)
See Also

The [X,Y,Z]DataSource properties of plot objects.
**Purpose**

Match regular expression

**Syntax**

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:

- `regexp('str', 'expr')`
- `[start end extents match tokens names] = regexp('str', 'expr')`
- `[v1 v2 ...] = regexp('str', 'expr', 'q1', 'q2', ...)`
- `[v1 v2 ...] = regexp('str', 'expr', 'q1', 'q2', ..., 'once')`

**Description**

The following descriptions apply to both `regexp` and `regexpi`:

- `regexp('str', 'expr')` returns a row vector containing the starting index of each substring of `str` that matches the regular expression string `expr`. If no matches are found, `regexp` returns an empty array. The `str` and `expr` arguments can also be cell arrays of strings. See the guidelines listed below under “Multiple Strings and Expressions”.

- `[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).

- `[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 <code>str</code> that matches <code>expr</code></td>
<td><code>start</code></td>
</tr>
<tr>
<td>2</td>
<td>Row vector containing the ending index of each substring of <code>str</code> that matches <code>expr</code></td>
<td><code>end</code></td>
</tr>
<tr>
<td>3</td>
<td>Cell array containing the starting and ending indices of each substring of <code>str</code> that matches a token in <code>expr</code></td>
<td><code>tokenExtents</code></td>
</tr>
</tbody>
</table>

2-1843
**regexp, regexpi**

### Return Values for Regular Expressions

<table>
<thead>
<tr>
<th>Default Order</th>
<th>Description</th>
<th>Qualifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Cell array containing the text of each substring of <code>str</code> that matches <code>expr</code></td>
<td>match</td>
</tr>
<tr>
<td>5</td>
<td>Cell array containing the text of each token captured by <code>regexp</code>.</td>
<td>tokens</td>
</tr>
<tr>
<td>6</td>
<td>Structure array containing the name and text of each named token captured by <code>regexp</code>. If there are no named tokens in <code>expr</code>, <code>regexp</code> returns a structure array with no fields. Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression <code>(?&lt;tokenname&gt;)</code>.</td>
<td>names</td>
</tr>
</tbody>
</table>

\[ v1 \ v2 \ldots \] = `regexp`(`'str'`, `'expr'`, `'q1'`, `'q2'`, ..., `'once'`) returns just the first match found. The keyword `once` must come last in the argument list. Output and qualifier arguments are not required.

`regexp` `'str'` `'expr'` `'q1'` `'q2'` ... `'once'` is the command syntax for this function. Only the `'str'` and `'expr'` arguments are required.

**Remarks**

**Multiple Strings and 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.

See “Regular Expressions” in the MATLAB documentation for a listing of all regular expression elements supported by MATLAB.

`regexp` does not support international character sets.
Examples

Example 1
Return a row vector of indices that match words that start with c, end with t, and contain one or more vowels between them. Make the matches insensitive to letter case (by using regexpi):

```matlab
str = 'bat cat can car COAT court cut ct CAT-scan';
regexpi(str, 'c[aeiou]+t')
ans =
   5    17    28    35
```

Example 2
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 =
   1    9
ans =
   1   11
ans =
   1    2    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

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

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>';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**
Enter a string containing two names, the first and last names being in a different order:

```matlab
str = sprintf('John Davis
Rogers, James')
str =
    John Davis
    Rogers, James
```

Create an expression that generates first and last name tokens, assigning the names first and last to the tokens. Call `regexp` to get the text and names of each token found:

```matlab
expr = ...
    '(?<first>\w+)\s+(?<last>\w+)|(?<last>\w+),\s+(?<first>\w+)';

[tokens names] = regexp(str, expr, 'tokens', 'names');
```

Examine the tokens cell array that was returned. The first and last name tokens appear in the order in which they were generated: first name–last name, then last name–first name:

```matlab
tokens{:}
ans =
    'John'    'Davis'
ans =
    'Rogers'    'James'
```

Now examine the names structure that was returned. First and last names appear in a more usable order:

```matlab
names(:,1)
ans =
    first: 'John'
    last: 'Davis'
```
**regexp, regexpi**

```matlab
codes(:,2)
ans =
    first: 'James'
    last: 'Rogers'
```

**See Also**
regexp, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi
**Purpose**
Replace string using regular expression

**Syntax**

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

**Description**

`s = 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`.

If `str` is a cell array of strings, then the `regexprep` return value `s` is always a cell array of strings having the same dimensions as `str`.

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 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`.

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 the section on “Tokens” and the example “Using Tokens in a Replacement String” in the External Interfaces documentation for information on using tokens.)

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

By default, `regexprep` replaces all matches and is case sensitive. You can use one or more
of the following options with `regexprep`. Separate each option in `optionlist` with a comma.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ignorecase'</td>
<td>Ignore the case of characters when matching <code>expr</code> to <code>str</code>.</td>
</tr>
<tr>
<td>'preservecase'</td>
<td>Ignore case when matching (as with 'ignorecase'), but override the case of replace characters with the case of corresponding characters in <code>str</code> when replacing.</td>
</tr>
<tr>
<td>'once'</td>
<td>Replace only the first occurrence of <code>expr</code> in <code>str</code>.</td>
</tr>
<tr>
<td>N</td>
<td>Replace only the Nth occurrence of <code>expr</code> in <code>str</code>.</td>
</tr>
</tbody>
</table>

**Remarks**

See “Regular Expressions” in the MATLAB documentation for a listing of all regular expression metacharacters supported by MATLAB.

`regexprep` does not support international character sets.

**Examples**

**Example 1**

Perform a case-sensitive replacement on words starting with `m` and ending with `y`:

```matlab
str = 'My flowers may bloom in May';
pat = 'm(\w*)y';
regexprep(str, pat, 'April')
ans =
    My flowers April bloom in May
```

Replace all words starting with `m` and ending with `y`, regardless of case, but maintain the original case in the replacement strings:

```matlab
regexprep(str, pat, 'April', 'preservecase')
ans =
    April flowers april bloom in April
```
Example 2

Replace all variations of the words 'walk up' using the letters following walk as a token. In the replacement string

```matlab
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';
regexprep(str, pat, 'ascend$1')
an =
    I ascend, they ascended, we are ascending.
```

Example 3

This example operates on a cell array of strings. It searches for consecutive matching letters (e.g., 'oo') and uses a common replacement value ('--') for all matches. The function returns a cell array of strings having the same dimensions as the input cell array:

```matlab
str = {'Whose woods these are I think I know.' ;
       'His house is in the village though;' ;
       'He will not see me stopping here' ;
       'To watch his woods fill up with snow.'};

a = regexprep(str, '(.)\1', '--', 'ignorecase')
a =
    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

See Also

regexp, regexpi, strfind, findstr, strmatch, strcmp, strcmpi, strncmp, strncmpi
**rehash**

**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 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, like the Editor.

`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 General Preferences.

See Also

addpath, clear, path, rmpath

Toolbox Path Caching
rem

Purpose
Remainder after division

Syntax
R = rem(X,Y)

Description
R = rem(X,Y) if Y ~= 0, returns X - n.*Y where n = fix(X./Y). If Y is not an integer and the quotient X./Y is within roundoff error of an integer, then n is that integer. By convention, rem(X,0) is NaN. The inputs X and Y must be real arrays of the same size, or real scalars.

Remarks
So long as operands X and Y are of the same sign, the statement rem(X,Y) returns the same result as does mod(X,Y). However, for positive X and Y,

rem(-X,Y) = mod(-X,Y) - Y

The rem function returns a result that is between 0 and sign(X)*abs(Y). If Y is zero, rem returns NaN.

See Also
mod
rename (ftp)

**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 (ftp), mput (ftp)
repmat

Purpose
Replicate and tile an array

Syntax
B = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p...])
repmat(A,m,n)

Description
B = repmat(A,m,n) creates a large matrix B consisting of an m-by-n tiling of copies of A. 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 (m-by-n-by-p-by-...) array composed of copies of A. A may be multidimensional.

repmat(A,m,n) when A is a scalar, produces an m-by-n matrix filled with A’s value. This can be much faster than a*ones(m,n) when m or n is large.

Examples
In this example, repmat replicates 12 copies of the second-order identity matrix, resulting in a “checkerboard” pattern.

B = repmat(ee(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
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1

The statement N = repmat(NaN,[2 3]) creates a 2-by-3 matrix of NaNs.
Purpose
Reset graphics object properties to their defaults

Syntax
reset(h)

Description
reset(h) resets all properties having factory defaults on the object identified by h. To see the list of factory defaults, use the statement

get(0, 'factory')

If h is a figure, MATLAB does not reset Position, Units, PaperPosition, and 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” for related functions
Reshape

**Purpose**
Reshape array

**Syntax**

\[
B = \text{reshape}(A,m,n) \\
B = \text{reshape}(A,m,n,p,...) \\
B = \text{reshape}(A,[m n p ...]) \\
B = \text{reshape}(A,...,[],...) \\
B = \text{reshape}(A,siz)
\]

**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 \( \text{prod(size}(A)) \).

- \( B = \text{reshape}(A,...,[],...) \) 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 \( \text{siz} \), 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.

\[
A = \\
\begin{bmatrix}
1 & 4 & 7 & 10 \\
2 & 5 & 8 & 11 \\
3 & 6 & 9 & 12 \\
\end{bmatrix}
\]

\[
B = \text{reshape}(A,2,6)
\]

\[
B = \\
\begin{bmatrix}
1 & 3 & 5 & 7 & 9 & 11 \\
2 & 4 & 6 & 8 & 10 & 12 \\
\end{bmatrix}
\]

\[
B = \text{reshape}(A,2,[])
\]
reshape

B =

1  3  5  7  9  11
2  4  6  8 10 12

See Also  

shiftdim, squeeze

The colon operator :
residue

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

\[
\frac{b(s)}{a(s)} = \frac{b_1s^m + b_2s^{m-1} + b_3s^{m-2} + \ldots + b_{m+1}}{a_1s^n + a_2s^{n-1} + a_3s^{n-2} + \ldots + a_{n+1}}
\]

where \( b_j \) and \( a_j \) are the \( j \)th elements of the input vectors \( b \) and \( a \).

[b,a] = 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

\[
\frac{r_j}{s-p_j} + \frac{r_{j+1}}{(s-p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s-p_j)^m}
\]

2-1860
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, `resid2` 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

$$b = \begin{bmatrix} 5 & 3 & -2 & 7 \end{bmatrix}$$
$$a = \begin{bmatrix} -4 & 0 & 8 & 3 \end{bmatrix}$$

and you can calculate the partial fraction expansion as

$$[r, p, k] = \text{residue}(b, a)$$

$$r =$$

-1.4167
-0.6653
1.3320
Now, convert the partial fraction expansion back to polynomial coefficients.

\[ [b, a] = \text{residue}(r, p, k) \]

\[ b = \begin{bmatrix} -1.2500 & -0.7500 & 0.5000 & -1.7500 \end{bmatrix} \]

\[ a = \begin{bmatrix} 1.0000 & 0.0000 & -2.0000 & -0.7500 \end{bmatrix} \]

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**

decov, poly, roots

**References**

Purpose
Restore the default search path

Syntax
restoredefaultpath
restoredefaultpath; matlabrc

Description
restoredefaultpath sets the search path to include only installed products from the MathWorks. Run restoredefaultpath if you are having problems with the search path. If restoredefaultpath seems to correct the problem, run savepath. Start 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 User Guide
rethrow

**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 the following character array 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>
</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.

**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
```

**See Also**
error, lasterror, lasterr, try, catch, dbstop
**Purpose**
Return to the 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
```matlab
cmap = rgb2hsv(M)
```

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.

```matlab
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” for related functions
Purpose
Plot colormap

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” for related functions
Purpose
Ribbon plot

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);
z = peaks(x,y);
ribbon(y,z)
colormap hsv
ribbon

See Also plot, plot3, surface, waterfall

“Polygons and Surfaces” for related functions
rmappdata

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
**Purpose**

Remove directory

**Graphical Interface**

As an alternative to the `rmdir` function, use the delete feature in the Current Directory browser.

**Syntax**

```matlab
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 `lasterr`). Here, `status` is 1 for success and is 0 for error, and `message`, `messageid`, and the `s` input argument are optional.

**Examples**

**Remove Empty Directory**

To remove `myfiles` from the current directory, where `myfiles` is empty, type

```matlab
rmdir('myfiles')
```

If the current directory is `matlabr13/work`, and `myfiles` is in `d:/matlabr13/work/project/`, use the relative path to `myfiles`

```matlab
rmdir('project/myfiles')
```

or the full path to `myfiles`

```matlab
rmdir('d:/matlabr13/work/project/myfiles')
```
**rmdir**

**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

```
  stat =
        1

  mess =
     ..
```
indicating `myfiles` and its contents were removed.
See Also  
cd, copyfile, delete, dir, error, fileattrib, filebrowser, lasterr, mkdir, movefile
rmdir (ftp)

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 structure fields

Syntax
s = rmfield(s,'field')
s = rmfield(s,FIELDS)

Description
s = rmfield(s,'field') removes the specified field from the structure array s.

s = rmfield(s,FIELDS) removes more than one field at a time when FIELDS is a character array of field names or cell array of strings.

See Also
fieldnames, setfield, getfield, isfield, orderfields, dynamic field names
**rmpath**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Remove directories from MATLAB search path</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graphical Interface</strong></td>
<td>As an alternative to the rmpath function, use the Set Path dialog box. To open it, select Set Path from the File menu in the MATLAB desktop.</td>
</tr>
</tbody>
</table>
| **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 unquoted 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
**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
    |       Uiojects
```
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.

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,

get(0,'CurrentFigure')

returns the empty matrix. Note, however, that gcf always returns a figure handle, and creates one if there are no figure objects.

DeleteFcn  string
This property is not used, because you cannot delete the root object.

Diary  on | {off}
Diary file mode. When this property is on, MATLAB maintains a file (whose name is specified by the DiaryFile property) that saves a copy of all keyboard input and most of the resulting output. See also the diary command.

DiaryFile  string
Diary filename. The name of the diary file. The default name is diary.

Echo  on | {off}
Script echoing mode. When Echo is on, MATLAB displays each line of a script file as it executes. See also the echo command.
Root Properties

**ErrorMessage** string

*Text of last error message.* This property contains the last error message issued by MATLAB.

**FixedWidthFontName** font name

*Fixed-width font to use for axes, text, and uicontrols whose FontName is set to FixedWidth.* MATLAB uses the font name specified for this property as the value for axes, text, and uicontrol FontName properties when their FontName property is set to FixedWidth. Specifying the font name with this property eliminates the need to hardcode font names in MATLAB applications and thereby enables these applications to run without modification in locales where non-ASCII character sets are required. In these cases, MATLAB attempts to set the value of FixedWidthFontName to the correct value for a given locale.

MATLAB application developers should not change this property, but should create axes, text, and uicontrols with FontName properties set to FixedWidth when they want to use a fixed-width font for these objects.

MATLAB end users can set this property if they do not want to use the preselected value. In locales where Latin-based characters are used, Courier is the default.

**Format** short | {shortE} | long | longE | bank | hex | + | rat

*Output format mode.* This property sets the format used to display numbers. See also the format command.

- short — Fixed-point format with 5 digits
- shortE — Floating-point format with 5 digits
- shortG — Fixed- or floating-point format displaying as many significant figures as possible with 5 digits
- long — Scaled fixed-point format with 15 digits
- longE — Floating-point format with 15 digits
- longG — Fixed- or floating-point format displaying as many significant figures as possible with 15 digits
- bank — Fixed-format of dollars and cents
- hex — Hexadecimal format
• + — Displays + and – symbols
• rat — Approximation by ratio of small integers

**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.

\[
\begin{align*}
    v &= \text{get(0,'MonitorPosition')} \\
    v &= \\
    x \ y \ \text{width} \ \text{height} \ % \ \text{Primary monitor} \\
    x \ y \ \text{width} \ \text{height} \ % \ \text{Secondary monitor}
\end{align*}
\]

Note that MATLAB sets the value of the ScreenSize property to the combined size of the monitors.
Root Properties

- **Parent** handle
  
  *Handle of parent object.* This property always contains the empty matrix, because the root object has no parent.

- **PointerLocation** \([x,y]\)
  
  *Current location of pointer.* A vector containing the \(x\)- and \(y\)-coordinates of the pointer position, measured from the lower left corner of the screen. You can move the pointer by changing the values of this property. The **Units** property determines the units of this measurement.

  This property always contains the instantaneous pointer location, even if the pointer is not in a MATLAB window. A callback routine querying the **PointerLocation** can get a different value than the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

- **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.

**ScreenSize** four-element rectangle vector (read only)

*Screen size.* A four-element vector,

\[
\text{[left, bottom, width, height]}
\]

that defines the display size. left and bottom are 0 for all *units* except pixels, in which case left and bottom are 1. width and height are the screen dimensions in units specified by the *Units* property.

**Selected** on | off

This property has no effect on the root level.

**SelectionHighlight** {on} | off

This property has no effect on the root level.

**ShowHiddenHandles** on | {off}

*Show or hide handles marked as hidden.* When set to on, this property disables handle hiding and exposes all object handles regardless of the setting of an object’s *HandleVisibility* property. When set to off, all objects so marked remain hidden within the graphics hierarchy.

**Tag** string

*User-specified object label.* The *Tag* property provides a means to identify graphics objects with a user-specified label. While it is not necessary to identify the root object with a tag (since its handle is always 0), you can use this property to store any string value that you can later retrieve using *set.*

**Type** string (read only)

Class of graphics object. For the root object, *Type* is always 'root'.

**UIContextMenu** handle

This property has no effect on the root level.

**Units** {pixels} | normalized | inches | centimeters | points | characters

*Unit of measurement.* This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower left
Root Properties

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

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

Description

\[ r = \text{roots}(c) \]
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 \( c_1 s^r + \ldots + c_n s + c_{n+1} \).

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, \text{roots} and \text{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
\[
p = \begin{bmatrix} 1 & -6 & -72 & -27 \end{bmatrix}
\]
The roots of this polynomial are returned in a column vector by
\[
r = \text{roots}(p)
\]

\[
r = \\
12.1229 \\
-5.7345 \\
-0.3884
\]

Algorithm

The algorithm simply involves computing the eigenvalues of the companion matrix:
\[
A = \text{diag(ones(n-1,1),-1)}; \\
A(1,:) = -c(2:n+1)./c(1); \\
eig(A)
\]

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 \( c \).
roots

See Also fzero, poly, residue
Purpose

Angle histogram

Syntax

rose(theta)
rose(theta,x)
rose(theta,nbins)
rose(axes_handles,...)
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.
ros

Example

Create a rose plot showing the distribution of 50 random numbers.

```matlab
theta = 2*pi*rand(1,50);
rose(theta)
```

See Also

compass, feather, hist, line, polar

“Histograms” 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
-52   -43    49    44  -599   411   208   208
-49    -8     8    59   208   208    99  -911
  29   -44    52   -23   208   208  -911    99
**Purpose**  
Rotate matrix 90°

**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
\[
X = \\
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]
rotated by 90 degrees is
\[
Y = \text{rot90}(X) \\
Y = \\
\begin{bmatrix}
3 & 6 & 9 \\
2 & 5 & 8 \\
1 & 4 & 7 \\
\end{bmatrix}
\]

**See Also**  
flipdim, fliplr, flipud
Purpose

Rotate object about a 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.

`rotate(h, \text{direction}, \alpha)` rotates the graphics object \( h \) by \( \alpha \) degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

`rotate(\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.
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” for related functions
Purpose

Rotate 3-D view using mouse

Syntax

```matlab
rotate3d on
rotate3d off
rotate3d
rotate3d(figure_handle,...)
rotate3d(axes_handle,...)
```

Description

- `rotate3d on` enables mouse-base rotation on all axes within the current figure.
- `rotate3d off` disables interactive axes rotation in the current figure.
- `rotate3d` toggles interactive axes rotation in the current figure.
- `rotate3d(figure_handle,...)` enables rotation within the specified figure instead of the current figure.
- `rotate3d(axes_handle,...)` enables rotation only in the specified axes.

Using `rotate3d`

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.

See Also

- `camorbit`, `rotate`, `view`
- Object Manipulation for related functions
**round**

**Purpose**
Round to nearest integer

**Syntax**
Y = round(X)

**Description**
Y = round(X) rounds the elements of X to the nearest integers. For complex X, the imaginary and real parts are rounded independently.

**Examples**
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]
a =
Columns 1 through 4
-1.9000  -0.2000  3.4000  5.6000
Columns 5 through 6
 7.0000  2.4000 + 3.6000i

round(a)
ans =
Columns 1 through 4
-2.0000   0  3.0000  6.0000
Columns 5 through 6
 7.0000  2.0000 + 4.0000i

**See Also**
ceil, fix, floor
Purpose

Reduced row echelon form

Syntax

\[ R = \text{rref}(A) \]

\[ [R, jb] = \text{rref}(A) \]

\[ [R, jb] = \text{rref}(A, \text{tol}) \]

Description

\( R = \text{rref}(A) \) produces the reduced row echelon form of \( A \) using Gauss Jordan elimination with partial pivoting. A default tolerance of \((\max(\text{size}(A)) \times \text{eps} \times \text{norm}(A, \infty))\) 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}.

Note

The demo \text{rrefmovie}(A) enables you to sequence through the iterations of the algorithm.

Examples

Use \text{rref} on a rank-deficient magic square:

\[ A = \text{magic}(4), \; R = \text{rref}(A) \]

\[
A = \\
\begin{bmatrix}
16 & 2 & 3 & 13 \\
5 & 11 & 10 & 8 \\
9 & 7 & 6 & 12 \\
4 & 14 & 15 & 1 \\
\end{bmatrix}
\]


Rref

R =

1  0  0  1
0  1  0  3
0  0  1  -3
0  0  0  0

See Also

inv, lu, rank
Purpose
Convert real Schur form to complex Schur form

Syntax
\[ [U, T] = \text{rsf2csf}(U, T) \]

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.

\[ [U, T] = \text{rsf2csf}(U, T) \] converts the real Schur form to the complex form.

Arguments \( U \) and \( T \) represent the unitary and Schur forms of a matrix \( A \), respectively, that satisfy the relationships: \( A = U^*T^*U \) and \( U^*U = \text{eye}(\text{size}(A)) \). See \text{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 \( A \).
\[
U =
\begin{bmatrix}
-0.4916 & -0.2756 - 0.4411i & 0.2133 + 0.5699i & -0.3428 \\
-0.4980 & -0.1012 + 0.2163i & -0.1046 + 0.2093i & 0.8001 \\
-0.6751 & 0.1842 + 0.3860i & -0.1867 - 0.3808i & -0.4260 \\
-0.2337 & 0.2635 - 0.6481i & 0.3134 - 0.5448i & 0.2466
\end{bmatrix}
\]
rsf2csf

\[ T = \begin{bmatrix}
4.8121 & -0.9697 + 1.0778i & -0.5212 + 2.0051i & -1.0067 \\
0 & 1.9202 + 1.4742i & 2.3355 & 0.1117 + 1.6547i \\
0 & 0 & 1.9202 - 1.4742i & 0.8002 + 0.2310i \\
0 & 0 & 0 & 1.3474
\end{bmatrix} \]

See Also  

schur
2 save

Purpose
Save workspace variables on 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', 'var1', 'var2', ...)
save('filename', '-struct', 's')
save('filename', '-struct', 's', 'f1', 'f2', ...)
save('-regexp', expr1, expr2, ...)
save('... ', 'format')
save filename var1 var2 ...

Description
save by itself stores all workspace variables in a binary format in the current directory in a file named **matlab.mat**. Retrieve the data with **load**. MAT-files are double-precision, binary, MATLAB format files. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs external to MATLAB.

save('filename') stores all workspace variables in the current directory in **filename.mat**. To save to another directory, use the full pathname for the filename. If **filename** is the special string **stdio**, the save command sends the data as standard output.

save('filename', 'var1', 'var2', ...) saves only the specified workspace variables in **filename.mat**. Use the * wildcard to save only those variables that match the specified pattern. For example, save('A*') saves all variables that start with A.

save('filename', '-struct', 's') saves all fields of the scalar structure s as individual variables within the file **filename**.

save('filename', '-struct', 's', 'f1', 'f2', ...) saves as individual variables only those structure fields specified (s.f1, s.f2, ...).
save\(('-\text{regexp}')\), expr1, expr2, ...) saves those variables that match any of the regular expressions expr1, expr2, etc.

save(..., 'format') enables you to make use of other data formats available with the save function. See the following table.

<table>
<thead>
<tr>
<th>Format</th>
<th>How Data Is Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>-append</td>
<td>The specified existing MAT-file, appended to the end. See Remarks, below.</td>
</tr>
<tr>
<td>-ascii</td>
<td>8-digit ASCII format</td>
</tr>
<tr>
<td>-ascii -double</td>
<td>16-digit ASCII format</td>
</tr>
<tr>
<td>-ascii -tabs</td>
<td>Delimits with tabs</td>
</tr>
<tr>
<td>-ascii -double -tabs</td>
<td>16-digit ASCII format, tab delimited</td>
</tr>
<tr>
<td>-mat</td>
<td>Binary MAT-file form (default)</td>
</tr>
<tr>
<td>-v4</td>
<td>A format that MATLAB Version 4 can open</td>
</tr>
<tr>
<td>-v6</td>
<td>A format that MATLAB Version 6 and earlier can open</td>
</tr>
</tbody>
</table>

save filename var1 var2 ... is the command form of the syntax.

Remarks

By default, MATLAB compresses the data it saves to MAT-files. MATLAB also uses Unicode character encoding when saving character data. Specify the -v6 option if you want to disable both of these features for a particular save operation. If you save data to a MAT-file that you intend to load using MATLAB Version 6 or earlier, then you must specify the -v6 option when saving.

To override the compression and Unicode setting for all of your MATLAB sessions, use the Preferences dialog box. Open the Preferences dialog and select General and then MAT-Files. To disable data compression and Unicode encoding, click Ensure backward compatibility (-v6). To turn these features back on, click Use default features (Unicode and compression). See “General Preferences for MATLAB” in the Desktop Tools and Development Environment documentation for more information.
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

**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`:

```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`:

```matlab
save('d:\mymfiles\june10','vol','temp','-ASCII')
```

**Example 4**
Save the fields of structure `s1` as individual variables rather than as an entire structure.

```matlab
s1.a = 12.7;  s1.b = {'abc', [4 5; 6 7]};  s1.c = 'Hello!';
save newstruct.mat -struct s1;
clear
```

Check what was saved to `newstruct.mat`:
whos -file newstruct.mat

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
<tr>
<td>b</td>
<td>1x2</td>
<td>158</td>
<td>cell array</td>
</tr>
<tr>
<td>c</td>
<td>1x6</td>
<td>12</td>
<td>char array</td>
</tr>
</tbody>
</table>

Grand total is 16 elements using 178 bytes

Read only the b field into the MATLAB workspace.

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:

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
Example 7

This example is similar to the last one, except that it saves one variable uncompressed, and then a second variable compressed to the same MAT-file. It then loads this data back into the MATLAB workspace:

```matlab
x = ones(3000);
y = uint32(rand(3000) * 100);
```

```matlab
save c1 x; save c1 y -compress -append;
```

```matlab
d = dir('c1.mat');
d.bytes
ans =
    20952950
```

```matlab
clear
load c1
whos
```

```
Name      Size                   Bytes  Class
x      3000x3000              72000000  double array
y      3000x3000              36000000  uint32 array

Grand total is 18000000 elements using 108000000 bytes
```

See Also
load, clear, diary, fprintf, fwrite, who, workspace
saveas

Purpose
Save figure or model using specified format

Syntax
saveas(h,'filename.ext')
saveas(h,'filename','format')

Description
saveas(h,'filename.ext') saves the figure or model 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.

<table>
<thead>
<tr>
<th>ext Values</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 models)</td>
</tr>
<tr>
<td>jpg</td>
<td>JPEG image (invalid for Simulink models)</td>
</tr>
<tr>
<td>m</td>
<td>MATLAB M-file (invalid for Simulink models)</td>
</tr>
<tr>
<td>pbm</td>
<td>Portable bitmap</td>
</tr>
<tr>
<td>pcx</td>
<td>Paintbrush 24-bit</td>
</tr>
<tr>
<td>pgm</td>
<td>Portable Graymap</td>
</tr>
<tr>
<td>png</td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>ppm</td>
<td>Portable Pixmap</td>
</tr>
<tr>
<td>tif</td>
<td>TIFF image, compressed</td>
</tr>
</tbody>
</table>

saveas(h,'filename','format') saves the figure or model 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 types supported by print. The print device types include the formats listed in the table of extensions above as well as additional file formats. Use an extension from the table above or from the list of device types supported by print. When using the print device type to specify format for saveas, do not use the prefixed -d.

Remarks

You can use open to open files saved using saveas with an m or fig extension. Other formats are not supported by open. The Save As dialog box you access from the figure window’s File menu uses saveas, limiting the file extensions to m and fig. The Export dialog box you access from the figure window’s File menu uses saveas with the format argument.

Examples

Example 1 – Specify File Extension

Save the current figure that you annotated using the Plot Editor to a file named pred_prey using the MATLAB fig format. This allows you to open the file pred_prey.fig at a later time and continue editing it with the Plot Editor.

```matlab
saveas(gcf,'pred_prey.fig')
```

Example 2 – Specify File Format but No Extension

Save the current figure, using Adobe Illustrator format, to the file logo. Use the ai extension from the above table to specify the format. The file created is logo.ai.

```matlab
saveas(gcf,'logo', 'ai')
```

This is the same as using the Adobe Illustrator format from the print devices table, which is -dill; use doc print or help print to see the table for print device types. The file created is logo.ai. MATLAB automatically appends the ai extension for an Illustrator format file because no extension was specified.

```matlab
saveas(gcf,'logo', 'ill')
```

Example 3 – Specify File Format and Extension

Save the current figure to the file 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 model 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
open, print

“Printing” for related functions
**saveobj**

**Purpose**
Save an object to a MAT-file

**Syntax**

```matlab
B = saveobj(A)
```

**Description**

`B = saveobj(A)` is called by the MATLAB `save` function when object `A` is saved to a MAT-file. This call executes the `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;
```

**See Also**

`save`, `load`, `loadobj`
savepath

Purpose
Save current MATLAB search path to pathdef.m file

Graphical Interface
As an alternative to the savepath function, use the Set Path dialog box. To open it, select Set Path from the File menu in the MATLAB desktop.

Syntax
savepath
savepath newfile

Description
savepath saves the current MATLAB search path to pathdef.m. It returns

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>If the file was saved successfully</td>
</tr>
<tr>
<td>1</td>
<td>If the save failed</td>
</tr>
</tbody>
</table>

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
Purpose

2-D scatter/bubble graph

Syntax

scat\(\text{ter}(X,Y,S,C)\)
scat\(\text{ter}(X,Y)\)
scat\(\text{ter}(X,Y,S)\)
scat\(\text{ter}(...,\text{markertype})\)
scat\(\text{ter}(...,\text{filled})\)
scat\(\text{ter}(...,\text{PropertyName},\text{propertyvalue})\)
scat\(\text{ter}(\text{axes\_handle},...)\)
\(h = \text{scat\_\text{ter}(...)}\)
\(h\lines = \text{scat\_\text{ter}(\text{\textquoteleft v6},...)}\)

Description

scat\(\text{ter}(X,Y,S,C)\) 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.

\(C\) determines the colors 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 \text{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).

scat\(\text{ter}(X,Y)\) draws the markers in the default size and color.

scat\(\text{ter}(X,Y,S)\) draws the markers at the specified sizes \((S)\) with a single color. This type of graph is also known as a bubble plot.

scat\(\text{ter}(...,\text{markertype})\) uses the marker type specified instead of ‘o’ (see LineSpec for a list of marker specifiers).

scat\(\text{ter}(...,\text{filled})\) fills the markers.

scat\(\text{ter}(...,\text{PropertyName},\text{propertyvalue})\) creates the scatter graph, applying the specified property settings. See scattergroup properties for a description of properties.
scatter

scatter(axes_handles,...) plots into the axes with handle axes_handle instead of the current axes (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.

Examples

load seamount
scatter(x,y,5,z)

See Also

scatter3, plot3

“Scatter/Bubble Plots” for related functions

See Triangulation and Interpolation of Scatter Data for related information.

See “Scattergroup Properties” for property descriptions
scatter3

Purpose

3-D scatter plot

Syntax

scatter3(X,Y,Z,S,C)
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(...,markertype)
scatter3(...,'filled')
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 the markers at the specified sizes (S) with 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.

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).
**scatter3**

**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)
```

![3D scatter plot example](image)

**See Also**
scatter, plot3

See “Scattergroup Properties” for property descriptions

“Scatter/Bubble Plots” for related functions
You can set and query graphics object properties using the set and get commands or the Property Editor (propertyeditor).

Note that you cannot define default property values for scattergroup objects.

See Plot Objects for information on scattergroup objects.

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 whether 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.
Scattergroup Properties

**ButtonDownFcn**  
string or function handle

*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over the scattergroup object.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

The expression executes in the MATLAB workspace.

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

**CData**  
vector, m-by-3 matrix, ColorSpec

*Color of markers.* When CData is a vector the same length as XData and YData, the values in CData are linearly mapped to the colors in the current colormap. When CData is a length(XData)-by-3 matrix, it specifies the colors of the markers as RGB values. CData can also be a color string (see ColorSpec for a list of color string specifiers).

**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.
Scattergroup Properties

**Children**
array of graphics object handles

*Children of the scattergroup object.* An array containing the handle of a patch object parented to the scattergroup 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 the stem Children property unless you set the Root ShowHiddenHandles property to on:

```matlab
set(0,'ShowHiddenHandles','on')
```

**Clipping**
{on} | off

*Clipping mode.* MATLAB clips scatter plots to the axes plot box by default. If you set Clipping to off, lines might be displayed outside the axes plot box.

**CreateFcn**
string or function handle

*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates a scattergroup object. You must specify the callback during the creation of the object. For example,

```matlab
scatter(x,y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other scattergroup properties. Setting this property on an existing scattergroup 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 the scattergroup object is deleted (e.g., this might happen when you issue a `delete` command on the scattergroup 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`. 
Scattergroup Properties

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

See the BeingDeleted property for related information.

**DisplayName**

string

*Label used by plot legends.* The legend and the plot browser use this text for labels for any scattergroup objects appearing in these legends.

**EraseMode**

{normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase scatter child objects (the patch used to construct the scatter graph). 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.
**Scattergroup Properties**

**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 operation 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 the scattergroup object.

- **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`.
Scattergroup Properties

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.

HitTest {on} | off
Selectable by mouse click. HitTest determines whether the scattergroup 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 line objects that compose the stem plot. If HitTest is off, clicking the stemseries object selects the object below it (which is usually the axes containing it).

HitTestArea on | {off}
Select scattergroup object on markers or area of scatter graph. This property enables you to select scattergroup objects in two ways:

• Select by clicking on scatter markers (default).
• Select by clicking anywhere in the extent of the scatter graph.

When HitTestArea is off, you must click the scatter markers to select the scattergroup object. When HitTestArea is on, you can select the scattergroup object by clicking anywhere within the extent of the scatter graph (i.e., anywhere within a rectangle that encloses all the scatter markers).

Interruptible {on} | off
Callback routine interruption mode. The Interruptible property controls whether a scattergroup object callback can be interrupted by callbacks invoked subsequently.
**Scattergroup Properties**

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 stem property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**LineWidth**

width in points (default: 0.5 points)

*Width of line that draws the edge of markers.* You can set the thickness of the lines used to draw the markers to a value in points.

**Marker**

character (see table)

*Marker symbol.* The `Marker` property specifies the type of markers that are displayed on the scatter graph. The following table shows supported markers.

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>o</td>
<td>Circle</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>.</td>
<td>Point</td>
</tr>
<tr>
<td>x</td>
<td>Cross</td>
</tr>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
</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* uses the *CData* property to determine the *MarkerEdgeColor*.

### 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 the figure color if the axes *Color* property is set to *none* (which is the factory default for axes objects).

### Parent

**handle of axes, hggroup, or hgtransform**

*Parent of scattergroup object.* This property contains the handle of the scattergroup object’s parent. The parent of a scattergroup object is the axes that contains it. You can reparent scattergroup objects to other axes, hggroup, or hgtransform objects.

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 the scattergroup object is selected.
Scattergroup Properties

SelectionHighlight {on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing selection handles on the scatter markers. When SelectionHighlight is off, MATLAB does not draw the handles.

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.

For example, you might create a stemseries object and set the Tag property:

```matlab
t = scatter(x,y,'Tag','scatter1')
```

When you want to access the scattergroup object, you can use `findobj` to find the scattergroup object’s handle. The following statement changes the MarkerFaceColor property of the object whose Tag is scatter1.

```matlab
set(findobj('Tag','scatter1'),'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.

```matlab
t = findobj(gca,'Type','hggroup');
```

UIContextMenu handle of a uicontextmenu object

Associate a context menu with the scattergroup object. Assign this property the handle of a uicontextmenu object created in the scattergroup object’s parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB
Scattergroup Properties

displays the context menu whenever you right-click over the scattergroup object.

**UserData array**
*User-specified data.* This property can be any data you want to associate with the scattergroup object (including cell arrays and structures). The scattergroup object does not set values for this property, but you can access it using the set and get functions.

**Visible** {on} | off
*Visibility of scattergroup object and its children.* By default, scattergroup object visibility is on. This means all children of the scattergroup object are visible unless the child object’s Visible property is set to off. Setting a scattergroup object’s Visible property to off also makes its children invisible.

**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, by default, 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)

Links YData to MATLAB variable. Set this property to a MATLAB variable that, by default, 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 of z-coordinates defining the scattergroup object. XData and YData must be the same length.

ZDataSource string (MATLAB variable)

Link ZData to MATLAB variable. Set this property to a MATLAB variable that, by default, 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.
Scattergroup Properties

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**

\[ T = \text{schur}(A) \]
\[ T = \text{schur}(A,\text{flag}) \]
\[ [U,T] = \text{schur}(A,\ldots) \]

**Description**

The `schur` command computes the Schur form of a matrix.

\[ T = \text{schur}(A) \] returns the Schur matrix \( T \).

\[ T = \text{schur}(A,\text{flag}) \] for real matrix \( A \), returns a Schur matrix \( T \) in one of two forms depending on the value of \( \text{flag} \):

- \( '\text{complex}' \) \( T \) is triangular and is complex if \( A \) has complex eigenvalues.
- \( '\text{real}' \) \( T \) has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. \( '\text{real}' \) 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.

\[ [U,T] = \text{schur}(A,\ldots) \] also returns a unitary matrix \( U \) so that \( A = U*T*U' \) and \( U'*U = \text{eye}(\text{size}(A)) \).

**Examples**

\( H \) is a 3-by-3 eigenvalue test matrix:

\[ H = \begin{bmatrix} -149 & -50 & -154 \\ 537 & 180 & 546 \\ -27 & -9 & -25 \end{bmatrix} \]

Its Schur form is

\[ \text{schur}(H) \]

\[ \text{ans} = \begin{bmatrix} 1.0000 & -7.1119 & -815.8706 \\ 0 & 2.0000 & -55.0236 \\ 0 & 0 & 3.0000 \end{bmatrix} \]
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**

`schur` uses LAPACK routines 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, DORGHR, 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>

**See Also**

eig, hess, qz, rsf2csf

**References**

### Purpose
Script M-files

### 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 an 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 \( \text{eps} \), because \( \pi \) is a floating-point approximation to the exact value of \( \pi \).

**Definition**
The secant can be defined as
\[
\sec(x) = \frac{1}{\cos(x)}
\]
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, eps, pi, secd, sech
## secd

### Purpose
Secant of an argument in degrees

### Syntax

\[ Y = \text{secd}(X) \]

### Description

\[ Y = \text{secd}(X) \] is the secant of the elements of \( X \), expressed in degrees. For odd integers \( n \), \( \text{secd}(n \times 90) \) is infinite, whereas \( \text{sec}(n \times \pi/2) \) is large but finite, reflecting the accuracy of the floating point value of \( \pi \).

### See Also
asecd, sec
Purpose          Hyperbolic secant

Syntax          \( Y = \text{sech}(X) \)

Description     The \( \text{sech} \) 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 \(-2\pi \leq x \leq 2\pi\).

\[
x = -2\pi:0.01:2\pi;
\text{plot}(x,\text{sech}(x)), \text{grid} \text{ on}
\]

Algorithm        \( \text{sech} \) uses this algorithm.

\[
\text{sech}(z) = \frac{1}{\cosh(z)}
\]

Definition       The secant can be defined as

\[
\text{sech}(z) = \frac{1}{\cosh(z)}
\]
sech

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(h,'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.

For example, this statement sets the ButtonDownFcn of the current axes to selectmoveresize:

    set(gca,'ButtonDownFcn','selectmoveresize')

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

See Also
The ButtonDownFcn of axes and uicontrol graphics objects
“Object Manipulation” for related functions
**semilogx, semilogy**

**Purpose**
Semilogarithmic plots

**Syntax**

- `semilogx(Y)`
- `semilogx(X1,Y1,...)`
- `semilogx(X1,Y1,LineSpec,...)`
- `semilogx(...,'PropertyName',PropertyValue,...)`
- `h = semilogx(...)`
- `hlines = semilogx('v6',...)`
- `semilogy(...)`
- `h = semilogy(...)`
- `hlines = semilogy('v6',...)`

**Description**

`semilogx` and `semilogy` plot data as logarithmic scales for the x- and y-axis, respectively, logarithmic.

`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.

`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`.

`semilogy(...)` creates a plot using a base 10 logarithmic scale for the y-axis and a linear scale for the x-axis.

`h = semilogx(...)` and `h = semilogy(...)` return a vector of handles to lineseries graphics objects, one handle per line.

2-1934
semilogx, semilogy

**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)`

**Examples**

Create a simple `semilogy` plot.

```matlab
x = 0:.1:10;
semilogy(x,10.^x)
```

**See Also**

`line`, `LineStyle`, `loglog`, `plot`

“Basic Plots and Graphs” for related functions
sendmail

**Purpose**  
Send e-mail message to list of e-mail addresses

**Syntax**  
`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 2-1937. 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

```matlab
setpref('Internet','E_mail','myaddress@example.com');
```
**Note** The sendmail function does not support email servers that require authentication.

**Examples**

**Example of sendmail with Two Attachments**

```javascript
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.

```javascript
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
```
**set**

**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(0, 'FactoryObjectTypePropertyName')
a = set(h, 'Default')
a = set(h, 'DefaultObjectTypePropertyName')
<cell array> = 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(0, 'FactoryObjectTypePropertyName') returns the possible values
of the named property for the specified object type, if the values are strings. The
argument FactoryObjectTypePropertyName is the word Factory concatenated
with the object type (e.g., axes) and the property name (e.g., CameraPosition).

2-1938
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.

Examples
Set the Color property of the current axes to blue.

    set(gca, 'Color', 'b')

Change all the lines in a plot to black.

    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.

    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 `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” for related functions
set (timer)

**Purpose**
Configure or display timer object properties

**Syntax**

```matlab
set(obj)
pool_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 (timer)

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)
```

View the possible values of the `ExecutionMode` property.

```matlab
set(t, 'ExecutionMode')
```

Set the value of a specific timer object property.

```matlab
set(t, 'ExecutionMode', 'FixedRate')
```

Set the values of several properties of the timer object.

```matlab
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.

```plaintext
set(t, {'StartDelay', 'Period'}, {30, 30})
```

See Also

timer, get
setappdata

Purpose
Set 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 a name and a value. value can be any type of data.

See Also
getappdata, isappdata, rmappdata

Purpose
Return the 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. The resulting vector is sorted in ascending order. In set theory terms, c = A - B. A and B can be cell arrays of strings.

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,:).

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**

Return the 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. The resulting vector is sorted in ascending order. In set theory terms, \( c = A - B \). A and B can be cell arrays of strings.

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 \( c = a(i,:) \).

**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 field of structure array

**Syntax**

\[ s = \text{setfield}(s, 'field', v) \]

\[ s = \text{setfield}(s, \{i,j\}, 'field', \{k\}, v) \]

**Description**

\[ s = \text{setfield}(s, 'field', v) \], where \( s \) is a 1-by-1 structure, sets the contents of the specified field to the value \( v \). This is equivalent to the syntax \( s.field = v \).

\[ s = \text{setfield}(s, \{i,j\}, 'field', \{k\}, v) \] sets the contents of the specified field to the value \( v \). 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.

**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 Technical Note 32236 for information about using dynamic field names versus the `getfield` and `setfield` functions.

**Examples**

Given the structure

\[ \text{mystr}(1,1).name = 'alice'; \]
\[ \text{mystr}(1,1).ID = 0; \]
\[ \text{mystr}(2,1).name = 'gertrude'; \]
\[ \text{mystr}(2,1).ID = 1; \]

You can change the name field of `mystr(2,1)` using

\[ \text{mystr} = \text{setfield}('mystr', \{2,1\}, 'name', 'ted'); \]
\[ \text{mystr}(2,1).name \]

\[ \text{ans} = \]

\[ \text{ted} \]

The following example sets fields of a structure using `setfield` with variable and quoted field names and additional subscripting arguments.

\[ \text{class} = 5; \quad \text{student} = 'John_Doe'; \]
grades_Doe = [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
grades = [];

grades = setfield(grades, {class}, student, 'Math', ...
{10, 21:30}, grades_Doe);

You can check the outcome using the standard structure syntax.

grades(class).John_Doe.Math(10, 21:30)

ans =

85 89 76 93 85 91 68 84 95 73

See Also
getfield, fieldnames, isfield, orderfields, rmfield, dynamic field names
### setstr

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Set string flag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>This MATLAB 4 function has been renamed <code>char</code> in MATLAB 5.</td>
</tr>
</tbody>
</table>
**Purpose**
Set exclusive OR of two vectors

**Syntax**
- `c = setxor(A, B)`
- `c = setxor(A, B, 'rows')`
- `[c, ia, ib] = setxor(...)`

**Description**
- `c = setxor(A, B)` returns the values that are not in the intersection of `A` and `B`. The resulting vector is sorted. `A` and `B` can be cell arrays of strings.

- `c = setxor(A, B, 'rows')`, when `A` and `B` are matrices with the same number of columns, returns the rows that are not in the intersection of `A` and `B`.

- `[c, ia, ib] = setxor(...)` also returns index vectors `ia` and `ib` such that `c` is a sorted combination of the elements `c = a(ia)` and `c = b(ib)` or, for row combinations, `c = a(ia,:)` and `c = b(ib,:)`.

**Examples**
```matlab
a = [-1 0 1 Inf -Inf NaN];
b = [-2 pi 0 Inf];
c = setxor(a, b)
```

```matlab
c =
   -Inf   -2.0000   -1.0000    1.0000    3.1416       NaN
```

**See Also**
`intersect`, `ismember`, `issorted`, `setdiff`, `union`, `unique`
### shading

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Set color shading properties</th>
</tr>
</thead>
</table>
| **Syntax**  | shading flat  
shading faceted  
shading interp |
| **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. |
| **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)  
sphere(16)  
axis square  
shading interp  
title('Interpolated Shading') |
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
shading

The `EdgeColor` and `FaceColor` properties for surface and patch graphics objects

“Color Operations” 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.

```matlab
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
**showplottool**

**Purpose**
Show or hide one of the figure plot tools

**Syntax**

```matlab
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.

**See Also**
`getplottool`, `plottools`
shrinkfaces

**Purpose**
Reduce the size of patch faces

**Syntax**

```matlab
shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p), shrinkfaces(fv)
nfv = shrinkfaces(f,v,sf)
[ nf , nv ] = shrinkfaces( ... )
```

**Description**

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, 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]);
ddaspect([1 1 1]); view(3); axis tight
title('Original')

figure
p2 = patch(shrinkfaces(fv,.3));
set(p2,'FaceColor','red','EdgeColor',[.5,.5,.5]);
ddaspect([1 1 1]); view(3); axis tight
title('After Shrinking')
```
See Also

isosurface, patch, reducevolume, daspect, view, axis

“Volume Visualization” 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 an 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 \(-\pi \leq x \leq \pi\).  
\[
x = -\pi:0.01:\pi;
\]
\[
plot(x,\sin(x)), grid on
\]

![Graph of sine function](image)

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

\[
\sin(x + iy) = \sin(x)cosh(y) + i\cos(x)sinh(y)
\]

\[
\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}
\]
**sin**

**Algorithm**  
*sin* uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

**See Also**  
asin, asinh, sind, sinh
Purpose
Sine of an 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
asind, sin
single

**Purpose**

Convert to single-precision

**Syntax**

`B = single(A)`

**Description**

`B = single(A)` converts the matrix `A` to single precision, returning that value in `B`. `A` can be any numeric object (such as a `double`). If `A` is already single precision, `single` has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The `single` class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are `reshape`, `size`, the relational operators, subscripted assignment, and subscripted reference.

You can define your own methods for the `single` class by placing the appropriately named method in an `@single` directory within a directory on your path.

**Examples**

```matlab
a = magic(4);
b = single(a);

whos
  Name      Size         Bytes  Class
  a         4x4            128  double array
  b         4x4             64  single array
```

**See Also**

`double`
**Purpose**
Hyperbolic sine of an 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 \(-5 \leq x \leq 5\).

\[
x = -5:0.01:5;
plot(x, \sinh(x)), grid on
\]
size

**Purpose**
Array dimensions

**Syntax**
d = size(X)
[m,n] = size(X)
m = size(X,dim)
[d1,d2,d3,...,dn] = size(X)

**Description**
d = size(X) returns the sizes of each dimension of array X in a vector d with ndims(X) elements.

[m,n] = size(X) returns the size of matrix X in separate variables m and n.

m = size(X,dim) returns the size of the dimension of X specified by scalar dim.

[d1,d2,d3,...,dn] = size(X) returns the sizes of the first n dimensions of array X in separate variables.

If the number of output arguments n does not equal ndims(X), then for:

n > ndims(X) size returns ones in the “extra” variables, i.e., outputs ndims(X)+1 through n.

n < ndims(X) dn contains the product of the sizes of the remaining dimensions of X, i.e., dimensions n+1 through ndims(X).

**Note** For a Java array, 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 rand(2,3,4) is 3.

```matlab
m = size(rand(2,3,4),2)
```

```
m =  
3
```

Here the size is output as a single vector.
```matlab
d = size(rand(2,3,4))
d = 
\begin{array}{ccc}
  2 & 3 & 4 \\
\end{array}
Here the size of each dimension is assigned to a separate variable.

```matlab
[m,n,p] = size(rand(2,3,4))
m = 
\begin{array}{c}
  2 \\
\end{array}
n = 
\begin{array}{c}
  3 \\
\end{array}
p = 
\begin{array}{c}
  4 \\
\end{array}
```

**Example 2.** If \( X = \text{ones}(3,4,5) \), then

```matlab
[d1,d2,d3] = size(X)
d1 = 
\begin{array}{c}
  3 \\
\end{array}
d2 = 
\begin{array}{c}
  4 \\
\end{array}
d3 = 
\begin{array}{c}
  5 \\
\end{array}
```

But when the number of output variables is less than \( \text{ndims}(X) \):

```matlab
[d1,d2] = size(X)
d1 = 
\begin{array}{c}
  3 \\
\end{array}
d2 = 
\begin{array}{c}
  20 \\
\end{array}
```

The “extra” dimensions are collapsed into a single product.

If \( n > \text{ndims}(X) \), the “extra” variables all represent singleton dimensions:

```matlab
[d1,d2,d3,d4,d5,d6] = size(X)
d1 = 
\begin{array}{c}
  3 \\
\end{array}
d2 = 
\begin{array}{c}
  4 \\
\end{array}
d3 = 
\begin{array}{c}
  5 \\
\end{array}
d4 = 
\begin{array}{c}
  1 \\
\end{array}
d5 = 
\begin{array}{c}
  1 \\
\end{array}
d6 = 
\begin{array}{c}
  1 \\
\end{array}
```

2-1965
size

See Also: exist, length, numel, whos
slice

Purpose

Volumetric slice plot

Syntax

slice(V,sx,sy,sz)
slice(X,Y,Z,V,sx,sy,sz)
slice(V,XI,YI,ZI)
slice(X,Y,Z,V,XI,YI,ZI)
slice(...,'method')
slice(axes_handle,...)
h = slice(...)

Description

slice displays orthogonal slice planes through volumetric data.

slice(V,sx,sy,sz) draws slices along the x, y, z directions in the volume V at the points in the vectors sx, sy, and sz. V is an m-by-n-by-p volume array containing data values at the default location X = 1:n, Y = 1:m, Z = 1:p. Each element in the vectors sx, sy, and sz defines a slice plane in the x-, y-, or z-axis direction.

slice(X,Y,Z,V,sx,sy,sz) draws slices of the volume V. X, Y, and Z are three-dimensional arrays specifying the coordinates for V. X, Y, and Z must be monotonic and orthogonally spaced (as if produced by the function meshgrid). The color at each point is determined by 3-D interpolation into the volume V.

slice(V,XI,YI,ZI) draws data in the volume V for the slices defined by XI, YI, and ZI. XI, YI, and ZI are matrices that define a surface, and the volume is evaluated at the surface points. XI, YI, and ZI must all be the same size.

slice(X,Y,Z,V,XI,YI,ZI) draws slices through the volume V along the surface defined by the arrays XI, YI, ZI.

slice(...,'method') specifies the interpolation method. 'method' is 'linear', 'cubic', or 'nearest'.

- linear specifies trilinear interpolation (the default).
- cubic specifies tricubic interpolation.
- nearest specifies nearest-neighbor interpolation.

slice(axes_handles,...) plots into the axes with handle axes_handle instead of the current axes (gca).
h = slice(...) returns a vector of handles to surface graphics objects.

Remarks
The color drawn at each point is determined by interpolation into the volume V.

Examples
Visualize the function

\[ v = xe^{(-x^2-y^2-z^2)} \]

over the range \(-2 \leq x \leq 2, -2 \leq y \leq 2, 2 \leq z \leq 2:\)

\[ [x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2); \]
\[ v = x.*exp(-x.^2-y.^2-z.^2); \]
\[ xslice = [-1.2,.8,2]; yslice = 2; zslice = [-2,0]; \]
\[ slice(x,y,z,v,xslice,yslice,zslice) \]
\[ colormap hsv \]

Slicing At Arbitrary Angles
You can also create slices that are oriented in arbitrary planes. To do this,
• Create a slice surface in the domain of the volume (surf, linspace).
• Orient this surface with respect to the axes (rotate).
• Get the XData, YData, and ZData of the surface (get).
• Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a for loop “passes” the plane through the volume along the z-axis.

```matlab
for i = -2:.5:2
    hsp = surf(linspace(-2,2,20),linspace(-2,2,20),zeros(20)+i);
    rotate(hsp,[1,-1,1],30)
    xd = get(hsp,'XData');
    yd = get(hsp,'YData');
    zd = get(hsp,'ZData');
    delete(hsp)
    slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries
    hold on
    slice(x,y,z,v,xd,yd,zd)
    hold off
    axis tight
    view(-5,10)
    drawnow
end
```

The following picture illustrates three positions of the same slice surface as it passes through the volume.
Slicing with a Nonplanar Surface

You can slice the volume with any surface. This example probes the volume created in the previous example by passing a spherical slice surface through the volume.

```matlab
[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
```
slice

```matlab
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” for related functions

Exploring Volumes with Slice Planes for more examples
Purpose
Smooth 3-D data

Syntax
\[
W = \text{smooth3}(V) \\
W = \text{smooth3}(V, 'filter') \\
W = \text{smooth3}(V, 'filter', \text{size}) \\
W = \text{smooth3}(V, 'filter', \text{size}, \text{sd})
\]

Description
\[
W = \text{smooth3}(V) \text{ smoothes the input data } V \text{ and returns the smoothed data in } W. \\
W = \text{smooth3}(V, 'filter') \text{ filter determines the convolution kernel and can be the strings} \\
\begin{itemize}
  \item 'gaussian'
  \item 'box' (default)
\end{itemize} \\
W = \text{smooth3}(V, 'filter', \text{size}) \text{ sets the size of the convolution kernel (default is } [3 \ 3 \ 3]). \text{ If size is scalar, then size is interpreted as } [\text{size}, \text{size}, \text{size}]. \\
W = \text{smooth3}(V, 'filter', \text{size}, \text{sd}) \text{ sets an attribute of the convolution kernel. When } filter \text{ is gaussian, } \text{sd} \text{ is the standard deviation (default is } .65). 
\]

Examples
This example smooths some random 3-D data and then creates an isosurface with end caps.
\[
\begin{verbatim}
rand('seed',0) 
data = rand(10,10,10); 
data = smooth3(data,'box',5); 
p1 = patch(isosurface(data,.5), ... 
    'FaceColor','blue','EdgeColor','none'); 
p2 = patch(isocaps(data,.5), ... 
    'FaceColor','interp','EdgeColor','none'); 
isonormals(data,p1) 
view(3); axis vis3d tight 
camlight; lighting phong
\end{verbatim}
\]
See Also

isocaps, isonormals, isosurface, patch

“Volume Visualization” 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(...)
```

**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 <code>A</code> is a ...</th>
<th><code>sort(A)</code> ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>Sorts the elements of <code>A</code>.</td>
</tr>
<tr>
<td>Matrix</td>
<td>Sorts each column of <code>A</code>.</td>
</tr>
<tr>
<td>Multidimensional array</td>
<td>Sorts <code>A</code> 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, real, logical, and character arrays are permitted. In addition, any of the numeric types 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`. If `dim` is a vector, `sort` works iteratively on the specified dimensions. Thus, `sort(A,[1 2])` is equivalent to `sort(sort(A,2),1)`.

`B = sort(...,mode)` sorts the elements in the specified direction, depending on the value of `mode`.

- `'ascend'` Ascending order (default)
- `'descend'` Descending order
sort

[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 n-by-n matrix, then
each column of IX is a permutation vector of the corresponding column of A,
such that

   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.

Examples

Example 1. This example sorts a matrix A in each dimension, and then sorts
it a third time, requesting an array of indices for the sorted result.

A = [ 3 7 5
       0 4 2 ];

sort(A,1)

ans =
     0     4     2
     3     7     5

sort(A,2)

ans =
     3     5     7
     0     2     4

[B,IX] = sort(A,2)

B =
     3     5     7
     0     2     4

IX =
     1     3     2
     1     3     2
Example 2. This example sorts each column of a matrix in descending order.

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

\[ \text{sort}(A,1,'\text{descend}') \]

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

This is equivalent to

\[ \text{sort}(A,'\text{descend}') \]

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

See Also: 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 as a group 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. For example, sortrows(A,[2 3]) sorts the rows of A by the second column, and where these are equal, further sorts by 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
Given the 5-by-5 string matrix,

A = ['one ';'two ';'three ';'four ';'five '];

The commands B = sortrows(A) and C = sortrows(A,1) yield

B =
 five  four
 four  five
 one  one
 three  two
 two  three

c =
 five  four
 four  five
 one  one
 three  two
 two  three

See Also
sort
sound

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.

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.

See Also
auread, auwrite, soundsc, wavread, wavwrite
Purpose
Scale data and play as sound

Syntax
soundsc(y,Fs)
soundsc(y)
soundsc(y,Fs,bits)
soundsc(y,...,slim)

Description
soundsc(y,Fs) sends the signal in vector y (with sample frequency Fs) to the speaker on PC and most UNIX platforms. The signal y is scaled to the range $-1.0 \leq y \leq 1.0$ before it is played, resulting in a sound that is played as loud as possible without clipping.

Note
The playback duration that results from setting Fs depends on the sound card you have installed. Most sound cards support sample frequencies of approximately 5-10 kHz to 44.1 kHz. Sample frequencies outside this range can produce unexpected results.

soundsc(y) plays the sound at the default sample rate or 8192 Hz.

soundsc(y,Fs,bits) plays the sound using bits number of bits/sample if possible. Most platforms support $\text{bits} = 8$ or $\text{bits} = 16$.

soundsc(y,...,slim), where slim = [slow shigh], maps the values in y between slow and shigh to the full sound range. The default value is slim = [min(y) max(y)].

Remarks
MATLAB supports all Windows-compatible sound devices.

See Also
auread, auwrite, sound, wavread, wavwrite
spalloc

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**

\[
\begin{align*}
S &= \text{sparse}(A) \\
S &= \text{sparse}(i,j,s,m,n,nzmax) \\
S &= \text{sparse}(i,j,s,m,n) \\
S &= \text{sparse}(i,j,s) \\
S &= \text{sparse}(m,n)
\end{align*}
\]

**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 \( m = \text{max}(i) \) and \( n = \text{max}(j) \). 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.
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, $A \times S$ is at least as sparse as $S$.

If you divide a sparse matrix $S$ by 0, for each entry of $S$ that is 0, the corresponding entry of $S/0$ is also 0. This differs from the usual MATLAB arithmetic rule that $0/0 = \text{NaN}$, which applies to nonsparse matrices.

S = sparse(1:n,1:n,1) generates a sparse representation of the n-by-n identity matrix. The same S results from $S = \text{sparse} (\text{eye}(n,n))$, 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, nnz, nonzeros, nzmax, spones, sprandn, sprandsym, spy

The sparfun directory
Purpose
Form least squares augmented system

Syntax
S = spaugment(A,c)

Description
S = spaugment(A,c) creates the sparse, square, symmetric indefinite matrix
S = [c*I A; A' 0]. The matrix S is related to the least squares problem
\[ \min \|b - A*x\| \]
by
\[ r = b - A*x \\
S * \left[ \frac{r}{c}; x \right] = \left[ b; 0 \right] \]
The optimum value of the residual scaling factor c, involves \( \min(\text{svd}(A)) \) and \( \|r\| \), 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
spconvert

Purpose
Import matrix from sparse matrix external format

Syntax
S = spconvert(D)

Description
spconvert is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. spconvert is the second step in the process:

1 Load an ASCII data file containing [i,j,v] or [i,j,re,im] as rows into a MATLAB variable.
2 Convert that variable into a MATLAB sparse matrix.

S = spconvert(D) converts a matrix D with rows containing [i,j,s] or [i,j,r,s] to the corresponding sparse matrix. D must have an nnz or nnz+1 row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form [m n 0] or [m n 0 0] anywhere in D can be used to specify size(S). If D is already sparse, no conversion is done, so spconvert can be used after D is loaded from either a MAT-file or an ASCII file.

Examples
Suppose the ASCII file uphill.dat contains

```
1 1 1.000000000000000
1 2 0.500000000000000
2 2 0.333333333333333
1 3 0.333333333333333
2 3 0.250000000000000
3 3 0.200000000000000
1 4 0.250000000000000
2 4 0.200000000000000
3 4 0.166666666666667
4 4 0.142857142857143
4 4 0.000000000000000
```

Then the statements

```
load uphill.dat
H = spconvert(uphill)
```
$$H = \begin{pmatrix}
1,1 & 1.0000 \\
1,2 & 0.5000 \\
2,2 & 0.3333 \\
1,3 & 0.3333 \\
2,3 & 0.2500 \\
3,3 & 0.2000 \\
1,4 & 0.2500 \\
2,4 & 0.2000 \\
3,4 & 0.1667 \\
4,4 & 0.1429 \\
\end{pmatrix}$$

recreate \texttt{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.
**spdiags**

**Purpose**
Extract and create sparse band and diagonal matrices

**Syntax**

```
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)
```

**Description**
The `spdiags` function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible:

- `[B,d] = 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`. `d` is a vector of length `p` whose integer components specify the diagonals in `A`.

- `B = spdiags(A,d)` extracts the diagonals specified by `d`.

- `A = spdiags(B,d,A)` replaces the diagonals specified by `d` with the columns of `B`. The output is sparse.

- `A = spdiags(B,d,m,n)` creates an `m`-by-`n` sparse matrix by taking the columns of `B` and placing them along the diagonals specified by `d`.

**Note**
If a column of `B` is longer than the diagonal it's replacing, `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`.

**Arguments**
The `spdiags` function deals with three matrices, in various combinations, as both input and output.

- **A**
  An `m`-by-`n` matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on `p` diagonals.

- **B**
  A `min(m,n)`-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`.

2-1986
Roughly, \( A, B, \) and \( d \) are related by

\[
\text{for } k = 1:p \\
B(:,k) = \text{diag}(A,d(k)) \\
\text{end}
\]

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.

**Examples**

**Example 1.** 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 2.** The second example is not square.

\[
A = \begin{bmatrix}
11 & 0 & 13 & 0 \\
0 & 22 & 0 & 24 \\
0 & 0 & 33 & 0 \\
41 & 0 & 0 & 44 \\
0 & 52 & 0 & 0 \\
0 & 0 & 63 & 0 \\
0 & 0 & 0 & 74
\end{bmatrix}
\]

Here \( m = 7, n = 4, \) and \( p = 3 \).

The statement \([B,d] = \text{spdiags}(A)\) produces \( d = [\cdot 3 \ 0 \ 2]' \) 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 \texttt{spdiags(B,d,7,4)} reproduces the original A.

**Example 3.** This example shows how \texttt{spdiags} creates the diagonals when the columns of B are longer than the diagonals they are replacing.

\[
B = \text{repmat}((1:6)',[1 7])
\]

\[
B = \\
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 & 6
\end{bmatrix}
\]

\[
d = [-4 -2 -1 0 3 4 5];
\]

\[
A = \text{spdiags}(B,d,6,6);
\]

\[
\text{full}(A)
\]

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

**See Also** \texttt{diag}

2-1988
### 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 \( I = \text{sparse} \left( \text{eye}(1000,1000) \right) \), 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**  
\[ f = \text{spfun}(\text{fun}, S) \]

**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 \( \text{fun} \) returns zero for some nonzero elements of \( S \)).

\[ f = \text{spfun}(\text{fun}, S) \]  
evaluates \( \text{fun}(S) \) on the nonzero elements of \( S \). You can specify \( \text{fun} \) as a function handle for an M-file or anonymous function.

**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

\[ S = \text{spdiags([1:4]',0,4,4) \]

\[
S = \\
(1,1) 1 \\
(2,2) 2 \\
(3,3) 3 \\
(4,4) 4 \\
\]

Because \( \text{fun} \) returns nonzero values for all nonzero element of \( S \), \( f = \text{spfun(@exp, S)} \) 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.

\[ \text{full}(\exp(S)) \]

\[
\text{ans} = \\
2.7183 1.0000 1.0000 1.0000 \\
1.0000 7.3891 1.0000 1.0000 \\
\]
1.0000 1.0000 20.0855 1.0000
1.0000 1.0000 1.0000 54.5982

See Also  function handle (@), anonymous functions
sph2cart

**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. \text{THETA}, \text{PHI}, and \text{R} must all be the same size. \text{THETA} and \text{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

\[
\begin{align*}
\theta &= \text{phi} \\
\phi &= \text{theta} \\
x &= r \cdot \cos(\phi) \cdot \cos(\theta) \\
y &= r \cdot \cos(\phi) \cdot \sin(\theta) \\
z &= r \cdot \sin(\phi)
\end{align*}
\]

**See Also**
cart2pol, cart2sph, pol2cart

2-1992
**Purpose**
Generate sphere

**Syntax**
```matlab
sphere
sphere(n)
[X,Y,Z] = sphere(...)
```

**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
```
sphere

See Also  
cylinder, axis equal

“Polygons and Surfaces” 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” for related functions
spline

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.

If Y is a vector, Y(j) is taken as the value to be matched at x(j), hence Y must have the same length as x, except in the case described in “Exceptions” (1). The not-a-knot end conditions are used.

If Y is a matrix or an N-dimensional array, Y(:,...,; j) is taken as the value to be matched at x(j), hence the last dimension of Y must equal length(x), except in the case described in “Exceptions” (2).

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. For information regarding the size of yy, see ppval.

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(min(x)) = Y(1)
   - df(max(x)) = Y(end)
2 If Y is a matrix or an N-dimensional array with size(Y,N) equal to length(x)+2, the following hold:
   - f(x(j)) matches the value Y(:,...,; j+1) for j=1:length(x)
   - Df(min(x)) matches Y(:,;...,1)
   - Df(max(x)) matches Y(:,;...,end)

Note You can also perform spline interpolation using the interp1 function with the command interp1(x,y,xx,'spline'). Note that while spline
performs interpolation on rows of an input matrix, 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.

```matlab
x = 0:10;
y = sin(x);
xx = 0:.25:10;
yy = spline(x,y,xx);
plot(x,y,'o',xx,yy)
```

**Example 2.** This illustrates the use of clamped or complete spline interpolation where end slopes are prescribed. Zero slopes at the ends of an interpolant to the values of a certain distribution are enforced.

```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);
```
Example 3. The two vectors

\[
\begin{align*}
  t &= 1900:10:1990; \\
  p &= [ 75.995 \ 91.972 \ 105.711 \ 123.203 \ 131.669 \ldots \\
      & \quad \quad 150.697 \ 179.323 \ 203.212 \ 226.505 \ 249.633 ];
\end{align*}
\]

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

\[
\begin{align*}
  x &= \pi*[0:.5:2]; \\
  y &= [0\ 1\ 0\ -1\ 0\ 1\ 0];
\end{align*}
\]
\begin{verbatim}
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),...,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(:,end) are used as endslopes.

Example 5. The following code generates sine and cosine curves, then samples
the splines over a finer mesh.

x = 0:.25:1;
Y = [sin(x); cos(x)];
xx = 0:.1:1;
YY = spline(x,Y,xx);
plot(x,Y(1,:),'.',xx,YY(1,:),'-'); hold on;
plot(x,Y(2,:),'.',xx,YY(2,:),':'); hold off;
\end{verbatim}
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. \texttt{spline} uses the functions \texttt{ppval}, \texttt{mkpp}, and \texttt{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  \texttt{interp1}, \texttt{ppval}, \texttt{mkpp}, \texttt{unmkpp}

Purpose
Replace nonzero sparse matrix elements with ones

Syntax
R = spones(S)

Description
R = spones(S) generates a matrix R with the same sparsity structure as S, but with 1's in the nonzero positions.

Examples
c = sum(spones(S)) is the number of nonzeros in each column.

r = sum(spones(S'))' is the number of nonzeros in each row.

sum(c) and sum(r) are equal, and are equal to nnz(S).

See Also
nnz, spalloc, spfun
spparms

Purpose
Set parameters for sparse matrix routines

Syntax
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 the approximate minimum degree ordering, colamd. 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 /.

'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.

Note Cholesky-based \ and / on symmetric positive definite matrices use symamd.
LU-based \ and / (UMFPACK) on square matrices use a modified colamd.
QR-based \ and / on rectangular matrices use colamd.

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>values(1)</td>
<td>'spumoni'</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>values(2)</td>
<td>'thr_rel'</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>values(3)</td>
<td>'thr_abs'</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>values(4)</td>
<td>'exact_d'</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>values(5)</td>
<td>'supernd'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>values(6)</td>
<td>'rreduce'</td>
<td>3.0</td>
<td>1.0</td>
</tr>
<tr>
<td>values(7)</td>
<td>'wh_frac'</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>values(8)</td>
<td>'autommd'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>values(9)</td>
<td>'autoamd'</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>values(10)</td>
<td>'piv_tol'</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>values(11)</td>
<td>'bandden'</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>values(12)</td>
<td>'umfpack'</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

Sparse \( A\backslash b \) on symmetric positive definite \( A \) uses symmmd and Cholesky-based solver. symmmd uses the parameter 'autoamd' to turn the preordering on or off.

Sparse \( A\backslash b \) on general square \( A \) uses UMFPACK and its modified colamd reordering. colamd does not use the parameters above, other than 'autoamd' which turns the preordering on or off, and 'piv_tol' which controls the pivot tolerance. UMFPACK also responds to 'spumoni', as do the majority of the built-in sparse matrix functions.

**See Also**

\\, chol, 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.

**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.

See Also
sprand, sprandsym
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`. Also known as maximum traversal, maximum assignment, and size of a maximum matching in the bipartite graph of `A`.

Always `sprank(A) >= rank(full(A))`, and in exact arithmetic `sprank(A) == rank(full(sprandn(A)))` with probability 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 a string

**Syntax**
```matlab
[s, errmsg] = sprintf(format, A, ...)
```

**Description**
`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.

**Format String**
The `format` argument is a string containing 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>%05.2d</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>
</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>
</tbody>
</table>
The `sprintf` function behaves like its ANSI C language namesake with these exceptions and extensions.

- If you use `sprintf` to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the `fix`, `floor`, `ceil`, or `round` functions to change the value in the double into a value that can be represented as an integer before passing it to `sprintf`.
- The following nonstandard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot; or &quot; (two single quotes)</td>
<td>Single quotation mark</td>
</tr>
<tr>
<td>%</td>
<td>Percent character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
</table>
| b | The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like `%bx`.
| t | The underlying C data type is a float rather than an unsigned integer. |

For example, to print a double value in hexadecimal use the format `%bx`.

- The `sprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of A (columnwise) until all the elements are used up. The function then continues in a similar manner through any additional matrix arguments.
- If %s is used to print part of a nonscalar double argument, the following behavior occurs:
  - 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.

```matlab
Str = [65 66 67 pi];
sprintf('%s %f', Str)
ans =
   ABC 3.141593
```

b 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).

```matlab
Str = [pi 65 66 67];
sprintf('%s %f %s', Str)
ans =
3.141593e+00 65.000000 BC
```

c 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>Display of Negative Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%e or %E</td>
</tr>
<tr>
<td>PC</td>
<td>0.0000000e+000</td>
</tr>
<tr>
<td>Others</td>
<td>-0.0000000e+00</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))/2)</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>
<tr>
<td><code>sprintf('%s','hello')</code></td>
<td>hello</td>
</tr>
<tr>
<td><code>sprintf('The array is %dx%d.',2,3)</code></td>
<td>The array is 2x3</td>
</tr>
</tbody>
</table>
| `sprintf('
')`                              | Line termination character on all platforms |

### See Also

int2str, num2str, sscanf

### References


spy

Purpose

Visualize sparsity pattern

Syntax

spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',markersize)

Description

spy(S) plots the sparsity pattern of any matrix S.

spy(S,markersize), where markersize is an integer, plots the sparsity pattern using markers of the specified point size.

spy(S,'LineSpec'), where LineSpec is a string, uses the specified plot marker type and color.

spy(S,'LineSpec',markersize) uses the specified type, color, and size for the plot markers.

S is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

Note

spy replaces format +, which takes much more space to display essentially the same information.

Examples

This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

B = bucky;
spy(B)
See Also

find, gplot, LineSpec, symamd, symmmd, 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
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, \(\text{norm}(A-X^2, \text{fro}')/\text{norm}(A, \text{fro}')\).

\[X, \alpha, \text{condest} = \text{sqrtm}(A)\] returns a stability factor \(\alpha\) and an estimate \(\text{condest}\) of the matrix square root condition number of \(X\). The residual \(\text{norm}(A-X^2, \text{fro}')/\text{norm}(A, \text{fro}')\) is bounded approximately by \(n*\alpha*\text{eps}\) and the Frobenius norm relative error in \(X\) is bounded approximately by \(n*\alpha*\text{condest}\*\text{eps}\), where \(n = \max(\text{size}(A))\).

Remarks

If \(X\) is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like \(X = [0 \ 0; \ 0 \ 0]\), 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

\[
X = \\
\begin{pmatrix}
5 & -4 & 1 & 0 & 0 \\
-4 & 6 & -4 & 1 & 0 \\
1 & -4 & 6 & -4 & 1 \\
0 & 1 & -4 & 6 & -4 \\
0 & 0 & 1 & -4 & 5
\end{pmatrix}
\]

This matrix is symmetric and positive definite. Its unique positive definite square root, \(Y = \text{sqrtm}(X)\), is a representation of the second difference operator.
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

\[
S = \begin{bmatrix}
-0.3723 & 0 \\
0 & -5.3723 \\
\end{bmatrix}
\]

All four \(Y\)s are of the form

\[
Y = V*S/V
\]
The \texttt{sqrtn} function chooses the two plus signs and produces $Y_1$, even though $Y_2$ is more natural because its entries are integers.

\textbf{See Also} \hspace{2em} \texttt{expm}, \texttt{funm}, \texttt{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 \( \text{size}(A, \text{dim}) = 1 \).

**Examples**
Consider the 2-by-1-by-3 array \( Y = \text{rand}(2,1,3) \). This array has a singleton column dimension — that is, there’s only one column per page.

\[
Y = \\
Y(:,:,1) = \begin{bmatrix} 0.5194 & 0.0346 \\ 0.8310 & 0.0535 \end{bmatrix} \\
Y(:,:,2) = \begin{bmatrix} 0.5297 \\ 0.6711 \end{bmatrix} \\
Y(:,:,3) = \begin{bmatrix} \end{bmatrix}
\]

The command \( Z = \text{squeeze}(Y) \) yields a 2-by-3 matrix:

\[
Z = \\
\begin{bmatrix} 0.5194 & 0.0346 & 0.5297 \\ 0.8310 & 0.0535 & 0.6711 \end{bmatrix}
\]

**See Also**
reshape, shiftdim
sscanf

**Purpose**
Read string under format control

**Syntax**

```matlab
A = sscanf(s, format)
A = sscanf(s, format, size)
[A, count, errmsg, nextindex] = sscanf(...)
```

**Description**

`A = sscanf(s, format)` reads data from the MATLAB string variable `s`, converts it according to the specified `format` string, and returns it in matrix `A`. `format` is a string specifying the format of the data to be read. See “Remarks” for details. `sscanf` is the same as `fscanf` except that it reads the data from a MATLAB string variable rather than reading it from a file.

`A = sscanf(s, format, size)` reads the amount of data specified by `size` and converts it according to the specified `format` string. `size` is an argument that determines how much data is read. Valid options are

<table>
<thead>
<tr>
<th><code>n</code></th>
<th>Read <code>n</code> elements into a column vector.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inf</code></td>
<td>Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.</td>
</tr>
<tr>
<td><code>[m,n]</code></td>
<td>Read enough elements to fill an <code>m</code>-by-<code>n</code> matrix, filling the matrix in column order. <code>n</code> can be <code>Inf</code>, but not <code>m</code>.</td>
</tr>
</tbody>
</table>

If the matrix `A` results from using character conversions only, and `size` is not of the form `[M,N]`, a row vector is returned.

`sscanf` differs from its C language namesakes `scanf()` and `fscanf()` in an important respect — it is vectorized in order to return a matrix argument. The `format` string is cycled through the file until an end-of-file is reached or the amount of data specified by `size` is read in.

`[A, count, errmsg, nextindex] = sscanf(...)` reads data from the MATLAB string variable `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 elements successfully read. `errmsg` is an optional output argument that returns an error message string if an error occurred or an empty matrix if an error did not occur. `nextindex` is an optional output argument specifying one more than the number of characters scanned in `s`. 

2-2023
When MATLAB reads a specified file, it attempts to match the data in the file to the format string. If a match occurs, the data is written into the matrix in column order. 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:

Add one or more of these characters between the % and the conversion character.

<table>
<thead>
<tr>
<th>An asterisk (*)</th>
<th>Skip over the matched value if the value is matched but not stored in the output matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A digit string</td>
<td>Maximum field width</td>
</tr>
<tr>
<td>A letter</td>
<td>The size of the receiving object; for example, h for short, as in %hd for a short integer, or 1 for long, as in %ld for a long integer or %lg for a double floating-point number</td>
</tr>
</tbody>
</table>

Valid conversion characters are as shown.

<table>
<thead>
<tr>
<th>%c</th>
<th>Sequence of characters; number specified by field width</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Decimal numbers</td>
</tr>
<tr>
<td>%e, %f, %g</td>
<td>Floating-point numbers</td>
</tr>
<tr>
<td>%i</td>
<td>Signed integer</td>
</tr>
<tr>
<td>%o</td>
<td>Signed octal integer</td>
</tr>
<tr>
<td>%s</td>
<td>A series of non-white-space characters</td>
</tr>
<tr>
<td>%u</td>
<td>Signed decimal integer</td>
</tr>
</tbody>
</table>
If %s is used, an element read might use several MATLAB matrix elements, each holding one character. Use %c to read space characters, or %s to skip all white space.

Mixing character and numeric conversion specifications cause the resulting matrix to be numeric and any characters read to appear as their ASCII values, one character per MATLAB matrix element.

For more information about format strings, refer to the scanf() and fscanf() routines in a C language reference manual.

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%x</td>
<td>Signed hexadecimal integer</td>
</tr>
<tr>
<td>[...</td>
<td>Sequence of characters (scanlist)</td>
</tr>
</tbody>
</table>

**Examples**

**Example 1**

The statements

```matlab
s = '2.7183 3.1416';
A = sscanf(s, '%f')
```

create a two-element vector containing poor approximations to e and pi.

**Example 2**

Create matrix A with both character and numeric data:

```matlab
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 fills matrix B in column order:

```matlab
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

**Syntax**

- `stairs(Y)`
- `stairs(X,Y)`
- `stairs(...,LineSpec)`
- `stairs(...,'PropertyName',propertyvalue)`
- `stairs(axes_handle,...)`
- `h = stairs(...)`
- `[xb,yb] = stairs(Y,...)`

**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. The elements of X must be monotonic.

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

\[ \text{length}(X) = \text{size}(Y,1) \]

`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_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (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 stastep plot of a sine wave.

```matlab
x = linspace(-2*pi,2*pi,40);
stairs(x,sin(x))
```

![Staircase plot example](image)

**See Also**

bar, hist, stem

“Discrete Data Plots” for related functions

See “Stairseries Properties” for property descriptions
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.
Stairseries Properties

**ButtonDownFcn**  
string or function handle  
*Button press callback function.* A callback that executes whenever you press a mouse button while the pointer is over the stairseries object.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

The expression executes in the MATLAB workspace.

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

**Children**  
array of graphics object handles  
*Children of the stairseries object.* An array containing the handles of all line objects parented to the stairseries 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 the `stairs` `Children` property unless you set the `Root/ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

**Clipping**  
{on} | off  
*Clipping mode.* MATLAB clips stairs plots to the axes plot box by default. If you set `Clipping` to `off`, lines might be displayed outside the axes plot box.

**Color**  
ColorSpec  
*Color of 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.

**CreateFcn**  
string or function handle  
*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates a stairseries object. You must specify the callback during the creation of the object. For example,

```
stairs(1:10,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.
MATLAB executes this routine after setting all other stairseries properties. Setting this property on an existing stairseries 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 the stairseries object is deleted (e.g., this might happen when you issue a delete command on the stairseries 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 and the plot browser use this text for labels for any stairseries objects appearing in these legends.

**EraseMode** {normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase stairs child 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 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 operation 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 the stairseries object.

- 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.

**HitTest**

`{on} | off`

*Selectable by mouse click.* `HitTest` determines if the `stairseries` 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 line objects that compose the stairs plot. If `HitTest` is `off`, clicking the `stairseries` object selects the object below it (which is usually the axes containing it).
Stairseries Properties

HitTestArea on | {off}

Select stairseries object on lines or area of extent. This property enables you to select stairseries objects in two ways:

- Select by clicking on lines (default).
- Select by clicking anywhere in the extent of the stairstep graph.

When HitTestArea is off, you must click the lines to select the stairseries object. When HitTestArea is on, you can select the stairseries object by clicking anywhere within the extent of the stairstep graph (i.e., anywhere within a rectangle that encloses all the stairstep graph).

Interruptible {on} | off

Callback routine interruption mode. The Interruptible property controls whether a stairseries object 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 stairs 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 used for the stairstep lines. Available line styles are shown in the table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>−−</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>−.</td>
<td>Dash-dot line</td>
</tr>
</tbody>
</table>
**Stairseries Properties**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<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 the stairs lines.* 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 the end of the stairs lines. 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>
</tbody>
</table>
Stairseries 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 stairs 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 the figure color, if the axes Color property is set to none (which is the factory default for axes).

**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 axes, hggroup, or hgtransform

*Parent of stairseries object.* This property contains the handle of the stairseries object’s parent. The parent of a stairseries 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 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 the stairseries object is selected.

**SelectionHighlight**  
{on} | off

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing selection handles on the stairseries object. 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 callbacks.

For example, you might create a stairseries object and set the Tag property:

```matlab
    t = stairs(Y, 'Tag', 'stairs1')
```

When you want to access the stairseries object, you can use `findobj` to find the stairseries object’s handle. The following statement changes the MarkerFaceColor property of the object whose Tag is stairs1.

```matlab
    set(findobj('Tag', 'stairs1'), 'MarkerFaceColor', '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.

```matlab
    t = findobj(gca, 'Type', 'hggroup');
```

**UIContextMenu**  
handle of a uicontextmenu object

*Associate a context menu with the stairseries object.* Assign this property the handle of a uicontextmenu object created in the stairseries object’s parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the stairseries object.
**UserData**

array

*User-specified data.* This property can be any data you want to associate with the stairseries object (including cell arrays and structures). The stairseries object does not set values for this property, but you can access it using the set and get functions.

**Visible**

{on} | off

*Visibility of stairseries object and its children.* By default, stairseries object visibility is on. This means all children of the stairs are visible unless the child object’s Visible property is set to off. Setting a stairseries object’s Visible property to off also makes its children invisible.

**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, \( \text{length}(\text{XData}) = \text{size}(\text{YData},1) \). XData must be monotonic.

If you do not specify XData (i.e., the input argument x), the stairs function uses the indices of YData to create the stairstep graph. See the XDataMode property for related information.

**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 stairs function sets this property to manual.

If you set XDataMode to auto after having specified XData, the stairs function resets the stairs locations and x tick-mark labels to the indices of the YData, overwriting any previous values.

**XDataSource**

string (MATLAB variable)

*Link XData to MATLAB variable.* Set this property to a MATLAB variable that, by default, 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.
Stairseries Properties

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.

**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, by default, is evaluated in the base workspace to generate the YData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change YData.

You can use the refreshdata function to force an update of the object’s data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

See the refreshdata reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Purpose
Start timer(s) running

Syntax
start(obj)

Description
start(obj) starts the timer running, represented by the timer object, obj. If obj is an array of timer objects, start starts all the timers. Use the timer function to create a timer object.

start sets the Running property of the timer object, obj, to 'on', initiates TimerFcn callbacks, and executes the StartFcn callback.

The timer stops running if one of the following conditions apply:

• The 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 the specified time

**Syntax**

\begin{verbatim}
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])
\end{verbatim}

**Description**

\texttt{startat(obj, time)} starts the timer running, represented by the timer object \texttt{obj}, at the time specified by the serial date number \texttt{time}. If \texttt{obj} is an array of timer objects, \texttt{startat} starts all the timers running at the specified time. Use the timer function to create the timer object.

\texttt{startat} sets the \texttt{Running} property of the timer object, \texttt{obj}, to 'on', initiates \texttt{TimerFcn} callbacks, and executes the \texttt{StartFcn} callback.

The serial date number, \texttt{time}, indicates the number of days that have elapsed since 1-Jan-0000 (starting at 1). See \texttt{datenum} for additional information about serial date numbers.

\texttt{startat(obj, S)} starts the timer running at the time specified by the date string \texttt{S}. The date string must use date format 0, 1, 2, 6, 13, 14, 15, 16, or 23, as defined by the \texttt{datestr} function. Date strings with two-character years are interpreted to be within the 100 years centered on the current year.

\texttt{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.

\texttt{startat(obj, Y, M, D)} \texttt{startat(obj, [Y, M, D])} start the timer at the year (\texttt{Y}), month (\texttt{M}), and day (\texttt{D}) specified. \texttt{Y}, \texttt{M}, and \texttt{D} must be arrays of the same size (or they can be a scalar).

\texttt{startat(obj, Y, M, D, H, MI, S)} \texttt{startat(obj, [Y, M, D, H, MI, S])} start the timer at the year (\texttt{Y}), month (\texttt{M}), day (\texttt{D}), hour (\texttt{H}), minute (\texttt{MI}), and second (\texttt{S}) specified. \texttt{Y}, \texttt{M}, \texttt{D}, \texttt{H}, \texttt{MI}, and \texttt{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

## 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 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 predefined aspects of MATLAB. See Startup Options and About Preferences in the MATLAB 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**
- \( s = \text{std}(X) \)
- \( s = \text{std}(X, \text{flag}) \)
- \( s = \text{std}(X, \text{flag}, \text{dim}) \)

**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. If \( X \) is a random sample of data from a normal distribution, \( s^2 \) is the best unbiased estimate of its variance.

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 \( X \).

- \( s = \text{std}(X, \text{flag}) \) for \( \text{flag} = 0 \), is the same as \( \text{std}(X) \). For \( \text{flag} = 1 \), \( \text{std}(X,1) \) returns the standard deviation using (2) above, producing the second moment of the sample about its mean.
s = std(X,flag,dim) computes the standard deviations along the dimension of X specified by scalar dim.

**Examples**
For matrix X

\[
X =
\begin{bmatrix}
1 & 5 & 9 \\
7 & 15 & 22
\end{bmatrix}
\]

s = std(X,0,1)
s =
\[
\begin{bmatrix}
4.2426 & 7.0711 & 9.1924
\end{bmatrix}
\]

s = std(X,0,2)
s =
\[
\begin{bmatrix}
4.000 \\
7.5056
\end{bmatrix}
\]

**See Also**
corrcoef, cov, mean, median
Purpose
Plot discrete sequence data

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_handles,...) plots into the axes with handle axes_handle instead of the current axes (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')
```

```
−8 −6 −4 −2 0 2 4 6 8
−1
−0.8
−0.6
−0.4
−0.2
0
0.2
0.4
0.6
0.8
1
```

```
−8 −6 −4 −2 0 2 4 6 8
−1
−0.8
−0.6
−0.4
−0.2
0
0.2
0.4
0.6
0.8
1
```
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 \( \exp(-0.07x) \cdot \cos(x) \).
- h(2) is the handle to the stemseries object plotting the expression \( \exp(0.05x) \cdot \cos(x) \).

```matlab
x = 0:25;
y = [\exp(-0.07*x).*cos(x);\exp(0.05*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.
stem

See Also  
bar, plot, stairs

See “Stemseries Properties” for property descriptions
Purpose

Plot three-dimensional discrete sequence data

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.

X = linspace(0,1,10);
Y = X./2;
Z = sin(X) + cos(Y);
stem3

stem3(X,Y,Z,'fill')
view(-25,30)

See Also

bar, plot, stairs, stem

“Discrete Data Plots” for related functions

“Stemseries Properties” for descriptions of properties.

Three-Dimensional Stem Plots for more examples.
**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 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 the stemseries object.

This property can be

- A string that is a valid MATLAB expression
- The name of an M-file
- A function handle

The expression executes in the MATLAB workspace.

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

### Children

`array of graphics object handles`

*Children of the stemseries object.* An array containing the handles of all line objects parented to the stemseries 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 the stem `Children` property unless you set the Root `ShowHiddenHandles` property to `on`:

```plaintext
set(0,'ShowHiddenHandles','on')
```
Stemseries Properties

**Stemseries Properties**

**Clipping**  
Clipping mode. MATLAB clips stem plots to the axes plot box by default. If you set Clipping to off, lines might be displayed outside the axes plot box.

**Color**  
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.

\[ h = \text{stem}(\text{randn}(10,1),'\text{Color}','r'); \]

**CreateFcn**  
string or function handle  
*Callback routine executed during object creation.* This property defines a callback that executes when MATLAB creates a stemseries object. You must specify the callback during the creation of the object. For example,

\[ \text{stem}(x,y,'\text{CreateFcnu}',@\text{CallbackFcn}) \]

where @CallbackFcn is a function handle that references the callback function.

MATLAB executes this routine after setting all other stemseries properties. Setting this property on an existing stemseries 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 the stemseries object is deleted (e.g., this might happen when you issue a delete command on the stemseries 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 and the plot browser use this text for labels for any stemseries objects appearing in these legends.

**EraseMode** \{normal\} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase stem child objects (the lines used to construct the stem plot). 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 operation 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 the stemseries object.

- **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
Stemseries Properties

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.

HitTest
{on} | off
Selectable by mouse click. HitTest determines whether the stemseries 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 line objects that compose the stem plot. If HitTest is off, clicking the stemseries object selects the object below it (which is usually the axes containing it).

HitTestArea
on | {off}
Select stemseries object on stem lines or area of extent. This property enables you to select stemseries objects in two ways:

- Select by clicking stem lines (default).
- Select by clicking anywhere in the extent of the stem graph.

When HitTestArea is off, you must click the stem lines (excluding the baseline) to select the stemseries object. When HitTestArea is on, you can select the stemseries object by clicking anywhere within the extent of the stem plot (i.e., anywhere within a rectangle that encloses all the stem lines).

Interruptible
{on} | off
Callback routine interruption mode. The Interruptible property controls whether a stemseries object 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.
Stemseries Properties

Setting `Interruptible` to `on` allows any graphics object’s callback to interrupt callback routines originating from a stem 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 used for the stem lines. Available line styles are shown in the table.

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

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`

*Width of the stem lines.* 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 the end of the stem lines. 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>
</tbody>
</table>
### Stemseries Properties

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

**MarkerEdgeColor**  
`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 stem `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 the figure color, if the axes `Color` property is set to `none` (which is the factory default for axes).
Stemseries Properties

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 axes, hggroup, or hgtransform

Parent of stemseries object. This property contains the handle of the stemseries object’s parent. The parent of a stemseries 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 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 the stemseries object has been selected.

SelectionHighlight  {on} | off

Objects are highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing selection handles on the stems. When SelectionHighlight is off, MATLAB does not draw the handles.

ShowBaseLine  {on} | off

Turn baseline display on or off. This property determines whether stem plots display a baseline from which the stems are drawn. By default, the baseline is displayed.

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')
```
**Stemseries Properties**

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.

```matlab
t = findobj(gca, 'Type', 'hggroup');
```

**UIContextMenu**

handle of a uicontextmenu object

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

**UserData**

array

*User-specified data.* This property can be any data you want to associate with the stemseries object (including cell arrays and structures). The stemseries object does not set values for this property, but you can access it using the `set` and `get` functions.

**Visible**

{on} | off

*Visibility of stemseries object and its children.* By default, stemseries object visibility is on. This means all children of the stem are visible unless the child object’s `Visible` property is set to `off`. Setting a stemseries object’s `Visible` property to `off` also makes its children invisible.

**XData**

array

*X-axis location of stems.* The stem function draws an individual stem at each x-axis location in the XData array. XData can be either a matrix equal in size to YData or a vector equal in length to the number of rows in YData. That is, `length(XData) == size(YData,1)`. XData does not need to be monotonically increasing.

If you do not specify XData (i.e., the input argument x), the stem function uses the indices of YData to create the stem plot. See the XDataMode property for related information.
Stemseries Properties

**XDataMode**

(auto) | manual

*Use automatic or user-specified x-axis values.* If you specify XData, MATLAB sets this property to manual.

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks and x-tick labels 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, by default, 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 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.

**YData**

scalar, vector, or matrix

*Stem plot data.* YData contains the data plotted as stems. Each value in YData is represented by a marker in the stem plot. If YData is a matrix, MATLAB creates a series of stems for each column in the matrix.

The input argument `y` in the `stem` function calling syntax assigns values to YData.

**YDataSource**

string (MATLAB variable)

*Link YData to MATLAB variable.* Set this property to a MATLAB variable that, by default, 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.
Stemseries Properties

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, by default, 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.
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Stop timer(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>stop(obj)</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>timer, start</td>
</tr>
</tbody>
</table>
str2double

Purpose
Convert string to double-precision value

Syntax
x = str2double('str')
X = str2double(C)

Description
X = str2double('str') converts the string str, which should be an ASCII character representation of a real or complex scalar value, to the MATLAB double-precision representation. The string can contain digits, a comma (thousands separator), a decimal point, a leading + or - sign, an e preceding a power of 10 scale factor, and an i for a complex unit.

If str does not represent a valid scalar value, str2double returns NaN.

X = str2double(C) converts the strings in the cell array of strings C to double precision. The matrix X returned will be the same size as C.

Examples
Here are some valid str2double conversions.

str2double('123.45e7')
str2double('123 + 45i')
str2double('3.14159')
str2double('2.7i - 3.14')
str2double({'2.71' '3.1415'})
str2double('1,200.34')

See Also
char, hex2num, num2str, str2num
**Purpose**
Construct a function handle from a function name string

**Syntax**
fhandle = 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 also perform this operation on a cell array of strings. In this case, an array of function handles is returned.

**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 makdHandle to construct the function handle:

```
makeHandle('sin')
ans =
    @sin
```

**Example 3**
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:

```
function myminbnd(fhandle, lower, upper)
    if ischar(fhandle)
        fhandle = str2func(fhandle);
    end
```

2-2067
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:

myminbnd('humps', 0.3, 1)
ans =
    0.6370

See Also function_handle, func2str, functions
Purpose
Form a blank padded character matrix from strings

Syntax
S = str2mat(T1, T2, T3, ...)

Description
S = str2mat(T1, T2, T3, ...) forms the matrix S containing the text strings T1, T2, T3, ... as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, Ti, can itself be a string matrix. This allows the creation of arbitrarily large string matrices. Empty strings are significant.

Note
This routine will become obsolete in a future version. Use char instead.

Remarks
str2mat differs from strvcat in that empty strings produce blank rows in the output. In strvcat, empty strings are ignored.

Examples
x = str2mat('36842', '39751', '38453', '90307');

whos x
Name     Size   Bytes  Class
x         4x5     40  char array

x(2,3)

ans =

7

See Also
cchar, strvcat
**Purpose**  
String to number conversion

**Syntax**  
\[ x = \text{str2num}('str') \]

**Description**  
\( x = \text{str2num}('str') \) converts the string \( str \), which is an ASCII character representation of a numeric value, to numeric representation. The 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.

The \text{str2num} function can also convert string matrices.

**Examples**  
\( \text{str2num}('3.14159e0') \) is approximately \( \pi \).

To convert a string matrix,

\[ \text{str2num}([ '1 2'; '3 4' ]) \]

\[ \begin{array}{cc}
1 & 2 \\
3 & 4 \\
\end{array} \]

**See Also**  
\text{num2str}, \text{hex2num}, \text{sscanf}, \text{sparse}, \text{special characters}
Strcat

**Purpose**

String concatenation

**Syntax**

\[ t = strcat(s1, s2, s3, \ldots) \]

**Description**

\[ t = strcat(s1, s2, s3, \ldots) \]

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 \([s1 \ s2 \ s3 \ \ldots]\) to preserve trailing spaces.

**Remarks**

`strcat` and matrix operation are different for strings that contain trailing spaces:

\[
\begin{align*}
a &= \text{'hello '} \\
b &= \text{'goodbye'} \\
\text{strcat(a, b)} \\
\end{align*}
\]

\[
\begin{align*}
an &= \text{hellogoodbye} \\
[a \ b] \\
an &= \text{hello goodbye}
\end{align*}
\]

**Examples**

Given two 1-by-2 cell arrays \( a \) and \( b \),

\[
\begin{align*}
a &= \text{'abcde'} \\
b &= \text{'fghi'} \\
\text{strcat(a, b)} \\
\end{align*}
\]

the command \( t = \text{strcat(a,b)} \) yields

\[
\begin{align*}
t &= \text{'abcdejkl'} \\
&\text{ 'fghimn'}
\end{align*}
\]

Given the 1-by-1 cell array \( c = \{ \text{Q} \} \), the command \( t = \text{strcat(a,b,c)} \) yields
strcat

t =
  'abcdejklQ'  'fghimnQ'

See Also  strvcat, cat, cellstr
**Purpose**

Compare strings

**Syntax**

\[
k = \	ext{strcmp}(\text{str1}', \text{str2}')
\]

\[
TF = \	ext{strcmp}(S, T)
\]

**Description**

\[
k = \	ext{strcmp}(\text{str1}', \text{str2}')\]

compares the strings \text{str1} and \text{str2} and returns logical true (1) if the two are identical and logical false (0) otherwise.

\[
TF = \	ext{strcmp}(S, T)\]

where either \( S \) or \( T \) is a cell array of strings, returns an array \( TF \) the same size as \( S \) and \( T \) containing 1 for those elements of \( S \) and \( T \) that match and 0 otherwise. \( S \) and \( T \) must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

**Remarks**

Note that the value returned by \text{strcmp} is not the same as the C language convention. In addition, the \text{strcmp} function is case sensitive; any leading and trailing blanks in either of the strings are explicitly included in the comparison.

\text{strcmp} is intended for comparison of character data. When used to compare numeric data, \text{strcmp} returns 0.

**Examples**

\[
\text{strcmp('Yes', 'No')} =
\begin{array}{l}
0 \\
\text{strcmp('Yes', 'Yes')} =
\begin{array}{l}
1 \\
A = \\
'\text{MATLAB}' \hspace{1cm} '\text{SIMULINK}' \\
'\text{Toolboxes}' \hspace{1cm} 'The MathWorks'
B = \\
'\text{Handle Graphics}' \hspace{1cm} 'Real Time Workshop' \\
'\text{Toolboxes}' \hspace{1cm} 'The MathWorks'
C = \\
'\text{Signal Processing}' \hspace{1cm} 'Image Processing' \\
'\text{MATLAB}' \hspace{1cm} '\text{SIMULINK}'
\end{array}
\end{array}
\]

\[
\text{strcmp}(A, B)\]

\[
\text{ans} =
\begin{array}{ll}
0 & 0 \\
1 & 1
\end{array}
\]
**strcmp**

```
strcmp(A, C)
ans =
  0   0
  0   0
```

**See Also**

strcmpi, strncmp, strncmpi, strmatch, strfind, findstr, regexp, regexpi, regexprep
### Purpose
Compare strings ignoring case

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcmpi(str1, str2)</code></td>
<td><code>strcmpi(S, T)</code> returns 1 if strings <code>str1</code> and <code>str2</code> are the same except for case and 0 otherwise.</td>
</tr>
<tr>
<td></td>
<td><code>strcmpi(S, T)</code> when either <code>S</code> or <code>T</code> is a cell array of strings, returns an array the same size as <code>S</code> and <code>T</code> containing 1 for those elements of <code>S</code> and <code>T</code> that match except for case, and 0 otherwise. <code>S</code> and <code>T</code> must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.</td>
</tr>
</tbody>
</table>

### Remarks
`strcmpi` is intended for comparison of character data. When used to compare numeric data, `strcmpi` returns 0.

`strcmpi` supports international character sets.

### See Also
`strcmp`, `strncmpi`, `strncmp`, `strmatch`, `strfind`, `findstr`, `regexp`, `regexpi`, `regexprep`
Purpose
Compute 2-D streamline data

Syntax
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)

Description
XY = stream2(x,y,u,v,startx,starty) computes streamlines from vector
data u and v. The arrays x and y define the coordinates for u and v and must be
monotonic and 2-D plaid (such as the data produced by meshgrid). startx and
starty define the starting positions of the streamlines. The section “Specifying
Starting Points for Stream Plots” provides more information on defining
starting points.

The returned value XY contains a cell array of vertex arrays.

XY = stream2(u,v,startx,starty) assumes the arrays x and y are defined as
[x,y] = meshgrid(1:n,1:m) where [m,n] = size(u).

XY = stream2(...,options) specifies the options used when creating the
streamlines. Define options as a one- or two-element vector containing the
step size or the step size and the maximum number of vertices in a streamline:

[stepsize]

or

[stepsize, max_number_vertices]

If you do not specify a value, MATLAB 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” for related functions

Specifying Starting Points for Stream Plots for related information
Purpose  Compute 3-D streamline data

Syntax  \[ XYZ = \text{stream3}(X,Y,Z,U,V,W,\text{startx},\text{starty},\text{startz}) \]
\[ XYZ = \text{stream3}(U,V,W,\text{startx},\text{starty},\text{startz}) \]

Description  \[ XYZ = \text{stream3}(X,Y,Z,U,V,W,\text{startx},\text{starty},\text{startz}) \] 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 \text{meshgrid}). \( \text{startx}, \text{starty}, \) and \( \text{startz} \) 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.

\[ XYZ = \text{stream3}(U,V,W,\text{startx},\text{starty},\text{startz}) \] assumes the arrays \( X, Y, \) and \( Z \) are defined as \( [X,Y,Z] = \text{meshgrid}(1:N,1:M,1:P) \) where \( [M,N,P] = \text{size}(U) \).

\[ XYZ = \text{stream3}(\ldots,\text{options}) \] 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:

\[ [\text{stepsize}] \]
or
\[ [\text{stepsize}, \text{max\_number\_vertices}] \]

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 \text{streamline} command to plot the data returned by \text{stream3}.

Examples  This example draws 3-D streamlines from data representing air currents over regions of North America.

\begin{verbatim}
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)
\end{verbatim}
See Also

coneplot, stream2, streamline

“Volume Visualization” for related functions

Specifying Starting Points for Stream Plots for related information
streamline

Purpose

Draw streamlines from 2-D or 3-D vector data

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:
streamline

[stepsize]

or

[stepsize, max_number_vertices]

If you do not specify values, MATLAB uses the default:

- Steplike = 0.1 (one tenth of a cell)
- Maximum number of vertices = 1000

streamline(axes_handles,...) plots into the axes with handle axes_handle instead of the current axes (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.

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” for related functions

Specifying Starting Points for Stream Plots for related information

Stream Line Plots of Vector Data for another example
Purpose
Display stream particles

Syntax
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)

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, then 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. Note that 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 streamparticles</th>
</tr>
</thead>
<tbody>
<tr>
<td>EraseMode</td>
<td>xor</td>
</tr>
<tr>
<td>LineStyle</td>
<td>none</td>
</tr>
<tr>
<td>Marker</td>
<td>o</td>
</tr>
<tr>
<td>MarkerEdgeColor</td>
<td>none</td>
</tr>
<tr>
<td>MarkerFaceColor</td>
<td>red</td>
</tr>
</tbody>
</table>

You can override any of these properties by specifying a property name and value as arguments to streamparticles. For example, this statement uses RGB values to set the MarkerFaceColor to medium gray:

    streamparticles(vertices,'MarkerFaceColor',[.5 .5 .5])

streamparticles(line_handle,...) uses the line object identified by line_handle to draw the stream particles.

    h = streamparticles(...) returns a vector of handles to the line objects it creates.

Examples

This example combines streamlines with stream particle animation. The interpstreamspeed function determines the vertices along the streamlines.
where stream particles will be drawn during the animation, thereby controlling the speed of the animation. Setting the axes DrawMode property to fast provides faster rendering.

load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,dy,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.

load wind
daspect([1 1 1]); view(2)
[verts averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
sl = streamline([verts averts]);
```matlab
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” for related functions

Creating Stream Particle Animations for more details

Specifying Starting Points for Stream Plots for related information
streamribbon

**Purpose**
Create a 3-D stream ribbon plot

**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

\[
[X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)
\]

where [m,n,p] = `size(U)`.

`streamribbon(vertices,X,Y,Z,cav,speed)` assumes precomputed streamline vertices, curl angular velocity, and flow speed. `vertices` is a cell array of streamline vertices (as produced by `stream3`). X, Y, Z, cav, and speed are 3-D arrays.

`streamribbon(vertices,cav,speed)` assumes X, Y, and Z are determined by the expression

\[
[X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p)
\]
where \([m,n,p] = \text{size}(\text{cav})\).

\text{streamribbon}(\text{vertices},\text{twistangle})\) uses the cell array of vectors \text{twistangle} for the twist of the ribbons (in radians). The size of each corresponding element of \text{vertices} and \text{twistangle} must be equal.

\text{streamribbon}(...,\text{width})\) sets the width of the ribbons to \text{width}.

\text{streamribbon}(\text{axes}\_\text{handles},...)\) plots into the axes with handle \text{axes}\_\text{handle} instead of the current axes (\text{gca}).

\(h = \text{streamribbon}(...)\) 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.

```matlab
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);
camlight; lighting gouraud
```
This example uses precalculated vertex data (stream3), curl average velocity (\( \text{curl} \)), 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.

```matlab
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*0.1;
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
% Define the twist angle
fullt = 0:.15:15;
verts = {[cos(fullt)' sin(fullt)' (fullt/3)']};
twistangle = {cos(fullt)'};
daspect([1 1 1])
streamribbon(verts,twistangle);
% Define viewing and lighting
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```
This example combines cone plots (coneplot) and stream ribbon plots in one graph.

```matlab
%-----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);
```
See Also

curl, streamtube, streamline, stream3

“Volume Visualization” for related functions

Displaying Curl with Stream Ribbons for another example

Specifying Starting Points for Stream Plots for related information
**streamslice**

**Purpose**
Draws streamlines in slice planes

**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(...,'arrowmode')
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.

You should 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 is 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] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = 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_handles,...) plots into the axes with handle
axes_handle instead of the current axes (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])
```
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.

```matlab
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” for related functions

Specifying Starting Points for Stream Plots for related information
Purpose

Creates a 3-D stream tube plot

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

\[ [X,Y,Z] = \text{meshgrid}(1:n,1:m,1:p) \]

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 \([m,n,p] = \text{size(divergence)}\).

\text{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.

\text{streamtube(vertices)}\) selects the width automatically.

\text{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 \(n = 20\).

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

\(h = \text{streamtube(...z)}\) 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
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
streamtube(verts,x,y,z,-div);
%-----Define viewing and lighting
view(3)
axis tight
shading interp
camlight; lighting gouraud
```
streamtube

See Also

divergence, streamribbon, streamline, stream3

“Volume Visualization” 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

\[ k = \text{strfind}(\text{str}, \text{pattern}) \]
\[ k = \text{strfind}([\text{cellstr} \text{, pattern}]) \]

Description

\[ k = \text{strfind}(\text{str}, \text{pattern}) \] searches the string \( \text{str} \) for occurrences of a shorter string, \( \text{pattern} \), and returns the starting index of each such occurrence in the double array \( k \). If \( \text{pattern} \) is not found in \( \text{str} \), or if \( \text{pattern} \) is longer than \( \text{str} \), then \text{strfind} returns the empty array \( [] \).

\[ k = \text{strfind}([\text{cellstr} \text{, pattern}]) \] searches each string in cell array of strings \( \text{cellstr} \) for occurrences of a shorter string, \( \text{pattern} \), and returns the starting index of each such occurrence in cell array \( k \). If \( \text{pattern} \) is not found in a string or if \( \text{pattern} \) is longer than all strings in the cell array, then \text{strfind} returns the empty array \( [] \), for that string in the cell array.

The search performed by \text{strfind} is case sensitive. Any leading and trailing blanks in \( \text{pattern} \) or in the strings being searched are explicitly included in the comparison.

Examples

Use \text{strfind} to find a two-letter pattern in string \( S \):

\[
S = 'Find the starting indices of the pattern string';
\]
\[
\text{strfind}(S, 'in')
\]
\[\begin{array}{c}
2 \\
15 \\
19 \\
45
\end{array}\]

\[
\text{strfind}(S, 'In')
\]
\[\begin{array}{c}
[]
\end{array}\]

\[
\text{strfind}(S, ' ')
\]
\[\begin{array}{c}
5 \\
9 \\
18 \\
26 \\
29 \\
33 \\
41
\end{array}\]

Use \text{strfind} on a cell array of strings:

\[
cstr = {'How much wood would a woodchuck chuck';
'if a woodchuck could chuck wood?'};
\]

\[
\text{idx} = \text{strfind}(cstr, 'wood');
\]
This means that 'wood' occurs at indices 10 and 23 in the first string and at indices 6 and 28 in the second.

See Also
findstr, strmatch, strtok, strcmp, strncmp, strcmpi, strncmpi, regexp, regexpi, regexprep
**Purpose**
MATLAB string handling

**Syntax**

\[
S = 'Any Characters' \\
S = \text{char}(X) \\
X = \text{double}(S)
\]

**Description**

\(S = 'Any Characters'\) creates a character array, or string. The string is actually a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The actual characters displayed depend on the character set encoding 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 \(S_1, S_2, \text{etc.}\) into a new character array, \(S\).

\(S = \text{strcat}(S_1, S_2, \ldots)\) concatenates \(S_1, S_2, \text{etc.}\), which can be character arrays or cell arrays of strings. When the inputs are all character arrays, the output is also a character array. When any of the inputs is a cell array of strings, \(\text{strcat}\) 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.

\(\text{ischar}(S)\) tells if \(S\) is a string variable. \(\text{iscellstr}(S)\) 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, cellstr, ischar, iscellstr, strvcat, sprintf, sscanf, input
**Purpose**

Justify a character array

**Syntax**

\[
\begin{align*}
T &= \text{strjust}(S) \\
T &= \text{strjust}(S, \ 'right') \\
T &= \text{strjust}(S, \ 'left') \\
T &= \text{strjust}(S, \ 'center')
\end{align*}
\]

**Description**

\[
\begin{align*}
T &= \text{strjust}(S) \text{ or } T = \text{strjust}(S, \ 'right') \text{ returns a right-justified version of the character array } S. \\
T &= \text{strjust}(S, \ 'left') \text{ returns a left-justified version of } S. \\
T &= \text{strjust}(S, \ 'center') \text{ returns a center-justified version of } S.
\end{align*}
\]

**See Also**

deblank
strmatch

Purpose
Find possible matches for a string

Syntax
\[ x = \text{strmatch('str', STRS)} \]
\[ x = \text{strmatch('str', STRS, 'exact')} \]

Description
\[ x = \text{strmatch('str', STRS)} \] looks through the rows of the character array or cell array of strings \( \text{STRS} \) to find strings that begin with string \( \text{str} \), returning the matching row indices. \text{strmatch} is fastest when \( \text{STRS} \) is a character array.

\[ x = \text{strmatch('str', STRS, 'exact')} \] returns only the indices of the strings in \( \text{STRS} \) matching \( \text{str} \) exactly.

Examples
The statement
\[ x = \text{strmatch('max', strvcat('max', 'minimax', 'maximum'))} \]
returns \( x = [1; 3] \) since rows 1 and 3 begin with 'max'. The statement
\[ x = \text{strmatch('max', strvcat('max', 'minimax', 'maximum'),'exact')} \]
returns \( x = 1 \), since only row 1 matches 'max' exactly.

See Also
\text{strcmp, strcmpi, strncmp, strncmpi, strfind, findstr, strvcat, regexp, regexpi, regexprep}
**strncmp**

**Purpose**
Compare the first \( n \) characters of two strings

**Syntax**

\[
k = \text{strncmp}('\text{str1}', '\text{str2}', n)\\
\text{TF} = \text{strncmp}(S, T, n)
\]

**Description**

\( k = \text{strncmp}('\text{str1}', '\text{str2}', n) \) returns logical true (1) if the first \( n \) characters of the strings \( \text{str1} \) and \( \text{str2} \) are the same, and returns logical false (0) otherwise. Arguments \( \text{str1} \) and \( \text{str2} \) can also be cell arrays of strings.

\( \text{TF} = \text{strncmp}(S, T, N) \), where either \( S \) or \( T \) is a cell array of strings, returns an array \( \text{TF} \) the same size as \( S \) and \( T \) containing 1 for those elements of \( S \) and \( T \) that match (up to \( n \) characters), and 0 otherwise. \( S \) and \( T \) must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

**Remarks**
The command \( \text{strncmp} \) is case sensitive. Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

\( \text{strncmp} \) is intended for comparison of character data. When used to compare numeric data, \( \text{strncmp} \) returns 0.

**See Also**

\( \text{strncmpi}, \text{strcmp}, \text{strcmpi}, \text{strmatch}, \text{strfind}, \text{findstr}, \text{regexp}, \text{regexpi}, \text{regexprep} \)
**strncmpi**

**Purpose**
Compare first $n$ characters of strings ignoring case

**Syntax**

```matlab
 strncmpi('str1', 'str2', n)
 TF = strncmpi(S, T, n)
```

**Description**

`strncmpi('str1', 'str2', n)` returns 1 if the first $n$ characters of the strings `str1` and `str2` are the same except for case, and 0 otherwise.

`TF = strncmpi(S, T, n)`, when either `S` or `T` is a cell array of strings, returns an array the same size as `S` and `T` containing 1 for those elements of `S` and `T` that match except for case (up to $n$ characters), and 0 otherwise. `S` and `T` must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

**Remarks**

`strncmpi` is intended for comparison of character data. When used to compare numeric data, `strncmpi` returns 0.

`strncmpi` supports international character sets.

**See Also**

`strncmp, strcmpi, strcmp, strmatch, strfind, findstr, regexp, regexp, regexp, regexp`
Purpose

Read formatted data from a string

Syntax

A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)

Description

A = strread('str') reads numeric data from input string str into a 1-by-N vector A, where N equals the number of whitespace-separated numbers in str. Use this form only with strings containing numeric data. See “Example 1” 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” 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” and “Example 5” below.

The table “Formats for strread” lists the valid format specifiers. More information on using formats is available under “Formats” 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” 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” 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” below.
## Formats for `strread`

<table>
<thead>
<tr>
<th>Format</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literals (ordinary characters)</td>
<td>Ignore the matching characters. For example, in a string that has <code>Dept</code> followed by a number (for department number), to skip the <code>Dept</code> and read only the number, use 'Dept' in the format string.</td>
<td>None</td>
</tr>
<tr>
<td>%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 string, which could be in double quotes.</td>
<td>Cell array of strings. Does not include the double quotes.</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>%*...</td>
<td>Ignore the characters following *. See Example 8 below.</td>
<td>No output</td>
</tr>
<tr>
<td>%w...</td>
<td>Read field width specified by w. The %f format supports %w.pf, where w is the field width and p is the precision.</td>
<td></td>
</tr>
</tbody>
</table>
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',';')
```
strread

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 reapplies the format specifiers, starting over at the beginning. See Example 5 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.

Examples

Example 1
Read numeric data into a 1-by-5 vector:
\[
a = \text{strread}('0.41 8.24 3.57 6.24 9.27')
a = \\
0.4100 8.2400 3.5700 6.2400 9.2700
\]

Example 2
Read numeric data into separate scalar variables:
\[
[a\ b\ c\ d\ e] = \text{strread}('0.41 8.24 3.57 6.24 9.27')
a = \\
a = 0.4100 \\
b = 8.2400 \\
c = 3.5700 \\
d = 6.2400 \\
e = 9.2700
\]

Example 3
Read the only first three numbers in the string, also formatting as floating point:
\[
a = \text{strread}('0.41 8.24 3.57 6.24 9.27', '%4.2f', 3)
\]
Example 4
Truncate the data to one decimal digit by specifying format %3.1f. The second specifier, %*1d, tells \texttt{strread} not to read in the remaining decimal digit:

\begin{verbatim}
    a = strread('0.41 8.24 3.57 6.24 9.27', '%3.1f %*1d')
\end{verbatim}

\begin{verbatim}
a =
    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
\end{verbatim}
Example 5
Read six numbers into two variables, reusing the format specifiers:

```
[a b] = strread('0.41 8.24 3.57 6.24 9.27 3.29', '%f %f')
```

```
a =
0.4100
3.5700
9.2700

b =
8.2400
6.2400
3.2900
```

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', ',')
```
Example 8
Read selected portions of the input string:
str = '<table border=5 width="100%" cellspacing=0>');?></strread(str, ...
   '%s%s %c %s "%4s" %s %c', 'delimiter', '= ')
border =
    5
width =
    '100%'
space =
    0

Example 9
Read the string into two vectors, restricting the Answer values to T and F. Also note that two delimiters (comma and space) are used here:
str = 'Answer_1: T, Answer_2: F, Answer_3: F';
[a b] = strread(str, '%s %[TF]', 'delimiter', ',', ', ')
a =
    'Answer_1: '
    'Answer_2: '
    'Answer_3: '
b =
    'T'
    'F'
    'F'

See Also    textread, sscanf
**Purpose**
String search and replace

**Syntax**
```
str = strrep(str1, str2, str3)
```

**Description**
```
str = strrep(str1, str2, str3) replaces all occurrences of the string str2 within string str1 with the string str3.

strrep(str1, str2, str3), when any of str1, str2, or str3 is a cell array of strings, returns a cell array the same size as str1, str2, and str3 obtained by performing a strrep using corresponding elements of the inputs. The inputs must all be the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the right number of rows.
```

**Examples**
```
s1 = 'This is a good example.,'
str = strrep(s1, 'good', 'great')
str =
This is a great example.

A =
'MATLAB'       'SIMULINK'
'Toolboxes'    'The MathWorks'

B =
'Handle Graphics'    'Real Time Workshop'
'Toolboxes'          'The MathWorks'

C =
'Signal Processing'    'Image Processing'
'MATLAB'               'SIMULINK'

strrep(A, B, C)
ans =
'MATLAB'       'SIMULINK'
'MATLAB'       'SIMULINK'
```

**See Also**
strfind
Purpose
First token in string

Syntax
\[
\text{token} = \text{strtok('str')}
\]
\[
\text{token} = \text{strtok('str', delimiter)}
\]
\[
\text{[token, rem]} = \text{strtok(...)}
\]

Description
token = strtok('str') uses the default delimiters, the white-space characters. These include tabs (ASCII 9), carriage returns (ASCII 13), and spaces (ASCII 32). Any leading white-space characters are ignored. If str is a cell array of strings, token is a cell array of tokens.

token = strtok('str', delimiter) returns the first token in the text string str, that is, the first set of characters before a delimiter is encountered. The vector delimiter contains valid delimiter characters. Any leading delimiters are ignored. If str is a cell array of strings, token is a cell array of tokens.

[token, rem] = strtok(...) returns the remainder rem of the original string. The remainder consists of all characters from the first delimiter on. If str is a cell array of strings, token is a cell array of tokens, and rem is a character array.

Examples
Example 1
This example uses the default white-space delimiter:

```matlab
s = ' This is a simple example.';
[token, rem] = strtok(s)
token =
This
rem =
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(''str'', delimiter)', ...
'<a name="13475"></a>token = strtok(''str'')');

rem = s;

while true
    [t{k}, rem] = strtok(rem, '<>');
    if isempty(t{k}), break; end
    disp(sprintf('%s',t{k}))
end

Here is the output:

ul class=continued
li class=continued
pre
a name="13474"
/a
token = strtok('str', delimiter)
a name="13475"
/a
token = strtok('str')

Example 3
Using strtok on a cell array of strings returns a cell array of strings in token
and a character array in rem:

s = {'all in good time'; ...
     'my dog has fleas'; ...
     'leave no stone unturned'};

rem = s;

for k = 1:4
    [token, rem] = strtok(rem);
token
end
Here is the output:

token =
   'all'
   'my'
   'leave'
token =
   'in'
   'dog'
   'no'
token =
   'good'
   'has'
   'stone'
token =
   'time'
   'fleas'
   'unturned'

**See Also**
findstr, strmatch
**Purpose**
Remove leading and trailing white-space from string

**Syntax**

S = strtrim(str)
C = strtrim(cstr)

**Description**

S = strtrim(str) returns a copy of string str with all leading and trailing white-space characters removed. A white-space character is one for which the isspace function returns 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('    \t 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
Purpose
Create structure array

Syntax
s = struct('field1', {}, 'field2', {}, ...)
s = struct('field1', values1, 'field2', values2, ...)

Description
s = struct('field1', {}, 'field2', {}, ...) creates an empty structure with fields field1, field2, ...

s = struct('field1', values1, 'field2', values2, ...) creates a structure array with the specified fields and values. The value arrays values1, values2, etc., must be cell arrays of the same size or scalar cells. Corresponding elements of the value arrays are placed into corresponding structure array elements. 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.

Remarks
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.

Examples
The command
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)
ans =
    type: 'big'
    color: 'red'
    x: 3

s(2)
ans =
  type: 'little'
  color: 'red'
  x: 4

Similarly, the command

```matlab
a.b = struct('z', {});
```

produces an empty structure `a.b` with field `z`.

```matlab
a.b
```

```matlab
ans =
  0x0 struct array with fields:
    z
```

See Also

- isstruct
- fieldnames
- isfield
- orderfields
- getfield
- setfield
- rmfield
- substruct
- deal
- cell2struct
- struct2cell
- dynamic field names
Purpose
Structure to cell array conversion

Syntax
c = struct2cell(s)

Description
c = struct2cell(s) converts the m-by-n structure s (with p fields) into a p-by-m-by-n cell array c.
If structure s is multidimensional, cell array c has size [p size(s)].

Examples
The commands

clear s, s.category = 'tree';
s.height = 37.4; s.name = 'birch';

create the structure

s =
   category: 'tree'
          height: 37.4000
         name: 'birch'

Converting the structure to a cell array,

   c = struct2cell(s)

   c =
      'tree'
   [37.4000]
     'birch'

See Also
   cell2struct, cell, iscell, struct, isstruct, fieldnames, dynamic field names
strvcat

Purpose
Vertical concatenation of strings

Syntax
S = strvcat(t1, t2, t3, ...)

Description
S = strvcat(t1, t2, t3, ...) forms the character array S containing the
text strings (or string matrices) t1, t2, t3, ... as rows. Spaces are appended to
each string as necessary to form a valid matrix. Empty arguments are ignored.

Remarks
If each text parameter, ti, is itself a character array, strvcat appends them
vertically to create arbitrarily large string matrices.

Examples
The command strvcat('Hello','Yes') is the same as ['Hello ';'Yes'],
except that strvcat performs the padding automatically.

    t1 = 'first'; t2 = 'string'; t3 = 'matrix'; t4 = 'second';
    S1 = strvcat(t1, t2, t3)    S2 = strvcat(t4, t2, t3)
    S1 =
    first
    string
    matrix
    S2 =
    second
    string
    matrix

    S3 = strvcat(S1, S2)
    S3 =
    first
    string
    matrix
    second
    string
    matrix

See Also
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 and control multiple axes

**Syntax**
```
subplot(m,n,p)
subplot(m,n,p,'replace')
subplot(m,n,p,'align')
subplot(h)
subplot('Position', [left bottom width height])
```

**Description**
`subplot` divides the current figure into rectangular panes that are numbered rowwise. Each pane contains an axes. Subsequent plots are output to the current pane.

- `subplot(m,n,p)` creates an axes in the pth pane of a figure divided into an m-by-n matrix of rectangular panes. The new axes becomes the current axes.

- If `p` is a vector, it specifies an axes having a position that covers all the subplot positions listed in `p`.

- `subplot(m,n,p,'replace')` If the specified axes already exists, delete it and create a new axes.

- `subplot(m,n,p,'align')` positions the individual axes so that the plot boxes align, but does not prevent the labels and ticks from overlapping.

- `subplot(h)` makes the axes 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.

**Remarks**
If a `subplot` specification causes a new axes to overlap any existing axes, then `subplot` deletes the existing axes and uicontrol objects. However, if the `subplot` specification exactly matches the position of an existing axes, then the matching axes 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

```matlab
subplot mnp
```

where m refers to the row, n refers to the column, and p specifies the pane.

**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, 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 in the default position. This syntax does not return a handle, so it is an error to specify a return argument. (This behavior is implemented 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.

```matlab
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.

```matlab
subplot(2,2,1:2)
subplot(2,2,3)
subplot(2,2,4)
```
See Also

axes, cla, clf, figure, gca

“Basic Plots and Graphs” for more information
**Purpose**

Overloaded method for \(A(i)=B\), \(A{I}=B\), and \(A.\text{field}=B\)

**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**

`subsasgn` is designed to be used by the MATLAB interpreter to handle indexed assignments to objects. Calling `subsasgn` directly as a function is not recommended. If you do use `subsasgn` in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.

**Examples**

The syntax \(A(1:2,:)=B\) calls \(A=\text{subsasgn}(A,S,B)\) 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 \(A=\text{subsasgn}(A,S,B)\) where \(S.type='{}'\).

The syntax \(A.\text{field}=B\) calls \(\text{subsasgn}(A,S,B)\) where \(S.type='.'\) and \(S.subs='\text{field}'\).

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases \(\text{length}(S)\) is the number of subscripting levels. For instance, \(A(1,2).\text{name}(3:5)=B\) calls \(A=\text{subsasgn}(A,S,B)\) where \(S\) is a 3-by-1 structure array with the following values:

<table>
<thead>
<tr>
<th>(S(1).\text{type}='()')</th>
<th>(S(2).\text{type}='.')</th>
<th>(S(3).\text{type}='()')</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S(1).\text{subs}={1,2})</td>
<td>(S(2).\text{subs}='\text{name}')</td>
<td>(S(3).\text{subs}={3:5})</td>
</tr>
</tbody>
</table>

**See Also**

`subsref`

See “Handling Subscripted Assignment” for more information about overloaded methods and `subsasgn`.
**subsindex**

**Purpose**
Overloaded method for $X(A)$

**Syntax**

```matlab
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 \( \text{acos}(A'\cdot B) \).

Remarks
If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations A, and a second realization of the experiment described by B, subspace(A,B) gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples
Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

\[
H = \text{hadamard}(8);
A = H(:,2:4);
B = H(:,5:8);
\]

Note that matrices A and B are different sizes—A has three columns and B four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

\[
\text{theta} = \text{subspace}(A,B)
\]
\[
\text{theta} = 1.5708
\]

That A and B are orthogonal is shown by the fact that theta is equal to \( \pi/2 \).

\[
\text{theta} - \text{pi}/2
\]
\[
\text{ans} = 0
\]
Purpose

Overloaded method for A(I), A{I} and A.field

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.

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

See “Handling Subscripted Reference” for more information about overloaded methods and subsref.
Purpose

Create structure argument for subsasgn or subsref

Syntax

S = substruct(type1,subs1,type2,subs2,...)

Description

S = substruct(type1,subs1,type2,subs2,...) creates a structure with the fields required by an overloaded subsref or subsasgn method. Each type string must be one of '.','()','{}'. 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 '.','()','{}
• subs: subscript values (field name or cell array of index vectors)

Examples

To call subsref with parameters equivalent to the syntax

B = A(3,5).field

you can use

S = substruct('()',{3,5},'.','field');
B = subsref(A,S);

The structure created by substruct in this example contains the following:

S(1)

ans =

    type: '()'
    subs: {[3] [5]}

S(2)

ans =

    type: '.'
    subs: 'field'

See Also

subsasgn, subsref
**sum**

**Purpose**
Sum of array elements

**Syntax**
- `B = sum(A)`
- `B = sum(A, dim)`
- `B = sum(A, 'double')`
- `B = sum(A, dim, 'double')`
- `B = sum(A, 'native')`
- `B = sum(A, 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`.
- `B = sum(..., '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')` 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
- `M = magic(3)`
- `M =
  8  1  6
  3  5  7
  4  9  2`

This is called a magic square because the sums of the elements in each column are the same.
as are the sums of the elements in each row, obtained by transposing:

\[
\text{sum}(M') = \\
15 \quad 15 \quad 15
\]

This section describes the support of \texttt{sum} for data types other than \texttt{double}.

**Data Type \texttt{single}**

You can apply \texttt{sum} to an array of type \texttt{single} and MATLAB returns an answer of type \texttt{single}. For example,

\[
\text{sum}(	ext{single}([2 5 8]))
\]

\[
\text{ans} = \\
15
\]

\[
\text{class(ans)}
\]

\[
\text{ans} = \\
\text{single}
\]

**Integer Data Types**

When you apply \texttt{sum} to any of the following integer data types, MATLAB returns an answer of type \texttt{double}:

- \texttt{int8} and \texttt{uint8}
- \texttt{int16} and \texttt{uint16}
- \texttt{int32} and \texttt{uint32}

For example,

\[
\text{sum}(	ext{single}([2 5 8]));
\]

\[
\text{class(ans)}
\]

\[
\text{ans} = \\
\]
sum

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)
```

```matlab
ans =
int8
```

See Also

cumsum, diff, prod, trac
**Purpose**

Extract subset of volume data set

**Syntax**

\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(X,Y,Z,V,\text{limits})
\]
\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(V,\text{limits})
\]
\[
Nv = \text{subvolume}(\ldots)
\]

**Description**

\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(X,Y,Z,V,\text{limits}) \text{ extracts a subset of the volume data set } V \text{ using the specified axis-aligned limits. limits} = \]
\[
[xmin,xmax,ymin,ymax,zmin,zmax] \text{ (Any NaNs in the limits indicate that the} \]
\[
\text{volume should not be cropped along that axis.)}
\]

The arrays \(X, Y, \) and \(Z\) define the coordinates for the volume \(V\). The subvolume is returned in \(NV\) and the coordinates of the subvolume are given in \(NX, NY, \) and \(NZ\).

\[
[Nx,Ny,Nz,Nv] = \text{subvolume}(V,\text{limits}) \text{ assumes the arrays } X, Y, \text{ and } Z \text{ are defined as}
\]
\[
[X,Y,Z] = \text{meshgrid}(1:N,1:M,1:P)
\]

where \([M,N,P] = \text{size}(V)\).

\[
Nv = \text{subvolume}(\ldots) \text{ returns only the subvolume.}
\]

**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).
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" for related functions
Superior class relationship

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 classes `class1`, `class2`, and so on.

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`
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Open MathWorks Technical Support Web page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>support</code></td>
</tr>
</tbody>
</table>
| **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

**Syntax**
surf(Z)
surf(X,Y,Z)
surf(X,Y,Z,C)
surf(...,'PropertyName',PropertyValue)
surf(axes_handle,...)
surfc(...)
h = surf(...)
h = surfc(...)
hsurface = surf('v6',...), hsurface = surfc('v6',...)

**Description**
Use `surf` and `surfc` to view mathematical functions over a rectangular region. `surf` and `surfc` create colored parametric surfaces specified by X, Y, and Z, with color specified by Z or C.

`surf(Z)` creates a 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(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,...)` plots 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',...) returns the handles of surface objects instead of surfaceplot objects for compatibility with MATLAB 6.5 and earlier.

**Algorithm**

Abstractly, a parametric surface is parametrized by two independent variables, i and j, which vary continuously over a rectangle; for example, $1 \leq i \leq m$ and $1 \leq j \leq n$. The three functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ specify the surface. When i and j are integer values, they define a rectangular grid with integer grid points. The functions $x(i,j)$, $y(i,j)$, and $z(i,j)$ become three m-by-n matrices, $X$, $Y$, and $Z$. Surface color is a fourth function, $c(i,j)$, denoted by matrix $C$.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
   i 1,j
   | i,j 1 i,j 1 i,j+1
   |   i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, $[X(:) Y(:) Z(:)]$ returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

Surface color can be specified in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of $x$ and $y$. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The shading function sets the shading. If the shading is **interp**, $C$ must be the same size as $X$, $Y$, and $Z$; it specifies the colors at the vertices. The color within
a surface patch is a bilinear function of the local coordinates. If the shading is faceted (the default) or flat, \( C(i,j) \) specifies the constant color in the surface patch:

\[
\begin{array}{cc}
(i,j) & (i,j+1) \\
| & C(i,j) | \\
(i+1,j) & (i+1,j+1)
\end{array}
\]

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.

**Examples**

Display a surface plot and contour plot of the \texttt{peaks} surface.

\[
\texttt{[X,Y,Z] = peaks(30);}
\]
\[
\texttt{surf(X,Y,Z)}
\]
\[
\texttt{colormap hsv}
\]
\[
\texttt{axis([-3 3 -3 3 -10 5])}
\]
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, mesh, pcolor, shading, trisurf, view

Properties for surfaceplot graphics objects

“Creating Surfaces and Meshes” 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**

\[
fvc = \text{surf2patch}(h)
\]
\[
fvc = \text{surf2patch}(Z)
\]
\[
fvc = \text{surf2patch}(Z,C)
\]
\[
fvc = \text{surf2patch}(X,Y,Z)
\]
\[
fvc = \text{surf2patch}(X,Y,Z,C)
\]
\[
fvc = \text{surf2patch}(...,'triangles')
\]
\[
[f,v,c] = \text{surf2patch}(...)
\]

**Description**

\[
fvc = \text{surf2patch}(h)\] 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.

\[
fvc = \text{surf2patch}(Z)\] calculates the patch data from the surface’s \( ZData \) matrix \( Z \).

\[
fvc = \text{surf2patch}(Z,C)\] calculates the patch data from the surface’s \( ZData \) and \( CData \) matrices \( Z \) and \( C \).

\[
fvc = \text{surf2patch}(X,Y,Z)\] calculates the patch data from the surface’s \( XData \), \( YData \), and \( ZData \) matrices \( X \), \( Y \), and \( Z \).

\[
fvc = \text{surf2patch}(X,Y,Z,C)\] calculates the patch data from the surface’s \( XData \), \( YData \), \( ZData \), and \( CData \) matrices \( X \), \( Y \), \( Z \), and \( C \).

\[
fvc = \text{surf2patch}(...,'triangles')\] creates triangular faces instead of the quadrilaterals that compose surfaces.

\[
[f,v,c] = \text{surf2patch}(...)\] returns the face, vertex, and color data in the three arrays \( f \), \( v \), and \( c \) instead of a struct.

**Examples**
The first example uses the 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 \text{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;
patch(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” for related functions
**Purpose**
Create surface object

**Syntax**
surface(Z)
surface(Z,C)
surface(X,Y,Z)
surface(X,Y,Z,C)
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.
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,

\[
\text{surface('XData',X,'YData',Y,'ZData',Z,'CData',C)}
\]

is equivalent to

\[
\text{surface(X,Y,Z,C)}
\]

When you specify only a single matrix input argument,

\[
\text{surface(Z)}
\]

MATLAB assigns the data properties as if you specified

\[
\text{surface('XData',[1:size(Z,2)],...}
\text{'YData',[1:size(Z,1)],...}
\text{'ZData',Z,...}
\text{'CData',Z)}
\]

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.

\[
\text{load clown}
\text{surface(peaks,flipud(X),...}
\text{'FaceColor','texturemap',...}
\]
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

Properties for surface graphics objects

“Creating Surfaces and Meshes” and “Object Creation Functions” for related functions
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 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**  
\[ m \times n \text{ matrix of } \text{double or uint8} \]

*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**  
\[ \text{none | direct | {scaled}} \]

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

- none — The transparency values of AlphaData are between 0 and 1 or are clamped to this range (the default).
- scaled — Transform the AlphaData to span the portion of the alphamap indicated by the axes ALim property, linearly mapping data values to alpha values.
• **direct** — use the **AlphaData** as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from 1 to \(\text{length(alphamap)}\). MATLAB maps values less than 1 to the first alpha value in the alphamap, and values greater than \(\text{length(alphamap)}\) 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 \(\text{uint8}\) 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  

*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
Surface Properties

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** string or function handle

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

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

**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 surface 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.
True color defines an RGB value for each vertex. If the coordinate data (XData, for example) are contained in m-by-n matrices, then CData must be an m-by-n-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

**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**

string or function handle

*Callback routine executed during object creation.* This property defines a callback routine that executes when MATLAB creates a surface object. You must define this property as a default value for surfaces or set the CreateFcn property during object creation.

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

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

MATLAB executes this routine after setting all surface properties. Setting this property on an existing surface object has no effect.
The handle of the object whose `CreateFcn` is being executed is 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 surface callback routine.* A callback routine that executes when you delete the surface object (e.g., when you issue a `delete` command or clear the axes or figure). 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 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`. 
Surface Properties

**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.

- **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.
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.
Surface Properties

**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

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

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See ColorSpec for more information on specifying color.
- **none** — Do not draw faces. Note that edges are drawn independently of faces.
- **flat** — The values of CData determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.
- **interp** — Bilinear interpolation of the values at each vertex (the CData) determines the coloring of each face.
- **texturemap** — Texture map the CData to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example.)

**FaceLighting**

\{none\} | flat | gouraud | phong

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

- **none** — Lights do not affect the faces of this object.
- **flat** — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
Surface Properties

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

**Interruptible**  
{on} | off

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

**LineStyle**  
{-} | -- | : | -. | none

*Edge line type.* This property determines the line style used to draw surface edges. The available line styles are shown in this table.

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

**LineWidth**

Scalar

*Edge line width.* The width of the lines in points used to draw surface edges. The default width is 0.5 points (1 point = 1/72 inch).

**Marker**

Marker symbol (see table)

*Marker symbol.* The Marker property specifies symbols that are displayed at vertices. You can set values for the Marker property independently from the LineStyle property.

You can specify these markers.

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

**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 surface (see ColorSpec for more information).

**MarkerSize**  size in points

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

**MeshStyle**  {both} | row | column

*Row and column lines.* This property specifies whether to draw all edge lines or just row or column edge lines.

- both draws edges for both rows and columns.
- row draws row edges only.
- column draws column edges only.

**NormalMode**  {auto} | manual

*MATLAB generated or user-specified normal vectors.* When this property is auto, MATLAB calculates vertex normals based on the coordinate data. If you
Surface Properties

specify your own vertex normals, MATLAB sets this property to manual and does not generate its own data. See also the VertexNormals property.

**Parent**
handle of axes, hggroup, or hgtransform

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

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

**Selected**
on | {off}

*Is object selected?* When this property is on, MATLAB displays a dashed bounding box around the surface if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

**SelectionHighlight**
{on} | off

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When SelectionHighlight is off, MATLAB does not draw the handles.

**SpecularColorReflectance**
scalar in the range 0 to 1

*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.
You can also set the intensity of the ambient and diffuse components of the light on the surface object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

**Tag**

`string`

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

**Type**

`string (read only)`

*Class of the graphics object.* The class of the graphics object. For surface objects, Type is always the string ‘surface’.

**UIContextMenu**

`handle of a uicontextmenu object`

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

**UserData**

`matrix`

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

**VertexNormals**

`vector or matrix`

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

**Visible**

`{on} | off`

*Surface object visibility.* By default, all surfaces are visible. When set to off, the surface is not visible, but still exists, and you can query and set its properties.
Surface 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 surface points. See the Description section for more information.
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.

This section lists property names along with the types of values each accepts. Curly braces {} enclose default values.

**AlphaData**  
An 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}

*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 surfaceplot 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 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.
Surfaceplot Properties

**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**
string or function handle

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

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

**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 surface plot 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.

True color defines an RGB value for each vertex. If the coordinate data (XData, for example) are contained in \(m\)-by-\(n\) matrices, then CData must be an \(m\)-by-\(n\)-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.
Surfaceplot Properties

CDataMapping {scaled} | direct

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

- **scaled** — Transform the color data to span the portion of the colormap indicated by the axes CLim property, linearly mapping data values to colors. See the caxis reference page for more information on this mapping.
- **direct** — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to \( \text{length}(\text{colormap}) \). MATLAB maps values less than 1 to the first color in the colormap, and values greater than \( \text{length}(\text{colormap}) \) to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

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 a surfaceplot object. You must specify the callback during the creation of the object. For example,

```
surf(peaks,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function. MATLAB executes this routine after setting all other surfaceplot properties. Setting this property on an existing surfaceplot 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 surfaceplot callback routine.* A callback routine that executes when you delete the surfaceplot object (e.g., when you issue a `delete` command or clear the axes or figure). 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`.
Surfaceplot Properties

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 surfaceplot object. See the AmbientStrength and SpecularStrength properties.

**EdgeAlpha**  
\{scalar = 1\} | flat | interp

*Transparency of the surfaceplot 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.
Surfaceplot Properties

- **flat** — The CData value of the first vertex for a face determines the color of each edge.

  ![Diagram](image)

  - Direction of increasing y data
  - Direction of increasing x data
  - Vertex controlling the color of adjacent edges

- **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 surfaceplot 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.
Surfaceplot Properties

- **none** — Do not erase the surfaceplot 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 surfaceplot by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the surfaceplot does not damage the color of the objects behind it. However, surfaceplot 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 surfaceplot 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 surfaceplot 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 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** — 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** — 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
Surfaceplot Properties

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 surfaceplot 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 surfaceplot 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 surfaceplot 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 surfaceplot edges. The available line styles are shown in this table.

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

LineWidth  scalar

Edge line width. The width of the lines in points used to draw surfaceplot 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.
Surfaceplot Properties

You can specify these markers.

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

**MarkerEdgeColor** none | {auto} | flat | ColorSpec

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

- none specifies no color, which makes nonfilled markers invisible.
- auto uses the same color as the EdgeColor property.
- flat uses the CData value of the vertex to determine the color of the marker edge.
- ColorSpec defines a single color to use for the edge (see ColorSpec for more information).
Surfaceplot Properties

**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 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 surfaceplot object.* This property contains the handle of the surfaceplot object’s parent. The parent of a surfaceplot 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.
Surfaceplot Properties

**Selected**
on | {off}

Is object selected? When this property is on, MATLAB displays a dashed bounding box around the surfaceplot 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

Intensity of specular light. This property sets the intensity of the specular component of the light falling on the surface. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surfaceplot object. See the AmbientStrength and DiffuseStrength properties. Also see the material function.

**Tag** string

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

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

**UserData**

matrix

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

**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

*Surfaceplot object visibility.* By default, all surfaceplots are visible. When set to off, the surfaceplot 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 surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of columns as ZData.

**XDataMode**

{auto} | manual

*Use automatic or user-specified x-axis values.* If you specify XData, MATLAB sets this property to manual.

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks and x-tick labels to the column indices of the ZData, overwriting any previous values for XData.
Surfaceplot 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 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.

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.

**ZData**
matrix
*Z-coordinates.* The z-position of the surfaceplot data points. See the Description section for more information.

**ZDataSource**
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 ZData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change ZData.

You can use the `refreshdata` function to force an update of the object’s data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

See the `refreshdata` reference page for more information.

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
Purpose
Surface plot with colormap-based lighting

Syntax
surfl(Z)
surfl(X,Y,Z)
surfl(...,'light')
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)

Description
The `surfl` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surfl(Z)` and `surfl(X,Y,Z)` create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. X, Y, and Z are vectors or matrices that define the x, y, and z components of a surface.

`surfl(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surfl(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surfl(...,s)` specifies the direction of the light source. s is a two- or three-element vector that specifies the direction from a surface to a light source. \( s = [sx \ sy \ sz] \) or \( s = [\text{azimuth} \ \text{elevation}] \). The default \( s \) is 45° counterclockwise from the current view direction.

`surfl(X,Y,Z,s,k)` specifies the reflectance constant. k is a four-element vector defining the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient. \( k = [ka \ kd \ ks \ shine] \) and defaults to \( [.55, .6, .4, 10] \).

\( h = \text{surfl}(...) \) 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 `surf1(X',Y',Z')`. Because of the way surface normal vectors are computed, `surf1` requires matrices that are at least 3-by-3.

**Examples**

View peaks using colormap-based lighting.

```matlab
[x,y] = meshgrid(3:1/8:3);
z = peaks(x,y);
surf1(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
surf1(peaks)
shading interp
colormap copper
hold off
```
surfl

See Also

colormap, shading, light

“Creating Surfaces and Meshes” for functions related to surfaces

“Lighting” for functions related to lighting
Purpose
Compute and display 3-D surface normals

Syntax

\[
\begin{align*}
\text{surfnorm}(Z) \\
\text{surfnorm}(X,Y,Z) \\
[Nx,Ny,Nz] = \text{surfnorm}(...) 
\end{align*}
\]

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.

\(\text{surfnorm}(Z)\) and \(\text{surfnorm}(X,Y,Z)\) 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.

\([Nx,Ny,Nz] = \text{surfnorm}(...)\) 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:

\[
\text{surfnorm}(X',Y',Z')
\]

`surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.

Algorithm
The surface normals are based on a bicubic fit of the data in \(X\), \(Y\), and \(Z\). For each vertex, diagonal vectors are computed and crossed to form the normal.

Examples
Plot the normal vectors for a truncated cone.

\[
\begin{align*}
[x,y,z] &= \text{cylinder}(1:10) \\
\text{surfnorm}(x,y,z) \\
\text{axis([-12 12 -12 12 -0.1 1])}
\end{align*}
\]
surf\text{norm}

\textbf{See Also} \quad \text{surf, quiver3}

“Colormaps” for related functions
**Purpose**
Singular value decomposition

**Syntax**

\[ s = \text{svd}(X) \]

\[ [U,S,V] = \text{svd}(X) \]

\[ [U,S,V] = \text{svd}(X,0) \]

**Description**
The `svd` command computes the matrix singular value decomposition.

\[ s = \text{svd}(X) \] returns a vector of singular values.

\[ [U,S,V] = \text{svd}(X) \] 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 \( X = U*S*V' \).

\[ [U,S,V] = \text{svd}(X,0) \] 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-\( n \).

**Examples**
For the matrix

\[
X = \\
1 & 2 \\
3 & 4 \\
5 & 6 \\
7 & 8
\]

the statement

\[ [U,S,V] = \text{svd}(X) \]

produces

\[
U = \\
-0.1525 & -0.8226 & -0.3945 & -0.3800 \\
-0.3499 & -0.4214 & 0.2428 & 0.8007 \\
-0.5474 & -0.0201 & 0.6979 & -0.4614 \\
-0.7448 & 0.3812 & -0.5462 & 0.0407
\]

\[
S = \\
14.2691 & 0 \\
0 & 0.6268
\]
svd

\[
\begin{pmatrix}
0 & 0 \\
0 & 0 \\
\end{pmatrix}
\]

\[
V = \\
-0.6414 & 0.7672 \\
-0.7672 & -0.6414 \\
\]

The economy size decomposition generated by

\[
[U, S, V] = \text{svd}(X, 0)
\]

produces

\[
U = \\
-0.1525 & -0.8226 \\
-0.3499 & -0.4214 \\
-0.5474 & -0.0201 \\
-0.7448 & 0.3812 \\
\]

\[
S = \\
14.2691 & 0 \\
0 & 0.6268 \\
\]

\[
V = \\
-0.6414 & 0.7672 \\
-0.7672 & -0.6414 \\
\]

**Algorithm**  
svd uses LAPACK routines to compute the singular value decomposition.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>DGESVD</td>
</tr>
<tr>
<td>Complex</td>
<td>ZGESVD</td>
</tr>
</tbody>
</table>

**Diagnostics**  
If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

**Solution will not converge.**

**References**  
svds

Purpose
A few singular values

Syntax
s = svds(A)
s = svds(A,k)
s = svds(A,k,0)
[U,S,V] = svds(A,...)

Description
svds(A) computes the five largest singular values and associated singular vectors of the matrix A.

svds(A,k) computes the k largest singular values and associated singular vectors of the matrix A.

svds(A,k,0) computes the k smallest singular values and associated singular vectors.

With one output argument, s is a vector of singular values. With 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

Algorithm
svds(A,k) uses eigs to find the k largest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0].

svds(A,k,0) uses eigs to find the 2k smallest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0], and then selects the k positive eigenvalues and their eigenvectors.

Example
west0479 is a real 479-by-479 sparse matrix. svd calculates all 479 singular values. svds picks out the largest and smallest singular values.

```
load west0479
s = svd(full(west0479))
s1 = svds(west0479,4)
ss = svds(west0479,6,0)
```
These plots show some of the singular values of *west0479* as computed by *svd* and *svds*.

The largest singular value of *west0479* can be computed a few different ways:

\[
\text{svds}(\text{west0479},1) = 3.189517598808622e+05 \\
\text{max(svd(full(west0479)))} = 3.18951759880862e+05 \\
\text{norm(full(west0479))} = 3.189517598808623e+05
\]

and estimated:

\[
\text{normest(west0479)} = 3.189385666549991e+05
\]

**See Also** svd, eigs
**switch**

**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
  ...
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 1 (true).

**Note for C Programmers**
Unlike the C language switch construct, the MATLAB switch does not “fall through.” That is, switch executes only the first matching case; subsequent matching cases do not execute. Therefore, break statements are not used.
Examples

To execute a certain block of code based on what the string, method, is set to,

```matlab
method = 'Bilinear';

switch lower(method)
    case {'linear','bilinear'}
        disp('Method is linear')
    case 'cubic'
        disp('Method is cubic')
    case 'nearest'
        disp('Method is nearest')
    otherwise
        disp('Unknown method.')
end

Method is linear
```

See Also

case, end, if, otherwise, while
Purpose

Symmetric approximate minimum degree permutation

Syntax

\[ p = \text{symamd}(S) \]
\[ p = \text{symamd}(S, \text{knobs}) \]
\[ [p, \text{stats}] = \text{symamd}(S) \]
\[ [p, \text{stats}] = \text{symamd}(S, \text{knobs}) \]

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.

\( \text{knobs} \) is a scalar. If \( S \) is \( n \)-by-\( n \), rows and columns with more than \( \text{knobs} \ast n \) entries are removed prior to ordering, and ordered last in the output permutation \( p \). If the \( \text{knobs} \) parameter is not present, then \( \text{knobs} = \text{spparms}('\text{whfrac}') \).

\( \text{stats} \) is an optional vector that provides data about the ordering and the validity of the matrix \( S \).

\[ \text{stats(1)} \quad \text{Number of dense or empty rows ignored by \text{symamd}} \]
\[ \text{stats(2)} \quad \text{Number of dense or empty columns ignored by \text{symamd}} \]
\[ \text{stats(3)} \quad \text{Number of garbage collections performed on the internal data structure used by \text{symamd}} \text{(roughly of size} \quad 8.4 \ast \text{nnz(tril}(S, -1)) + 9n \text{ integers)} \]
\[ \text{stats(4)} \quad 0 \text{ if the matrix is valid, or 1 if invalid} \]
\[ \text{stats(5)} \quad \text{Rightmost column index that is unsorted or contains duplicate entries, or 0 if no such column exists} \]
\[ \text{stats(6)} \quad \text{Last seen duplicate or out-of-order row index in the column index given by \text{stats(5)}, or 0 if no such row index exists} \]
\[ \text{stats(7)} \quad \text{Number of duplicate and out-of-order row indices} \]
Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to \texttt{symamd}. For this reason, \texttt{symamd} verifies that \( S \) is valid:

- If a row index appears two or more times in the same column, \texttt{symamd} ignores the duplicate entries, continues processing, and provides information about the duplicate entries in \texttt{stats}(4:7).
- If row indices in a column are out of order, \texttt{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 \texttt{stats}(4:7).
- If \( S \) is invalid in any other way, \texttt{symamd} cannot continue. It prints an error message, and returns no output arguments (\( p \) or \( \text{stats} \)).

The ordering is followed by a symmetric elimination tree post-ordering.

\textbf{Note} \texttt{symamd} tends to be faster than \texttt{symmmd} and tends to return a better ordering.

\textbf{See Also} \texttt{colamd, colmmd, colperm, spparms, symmmd, symrcm}

\textbf{References} The authors of the code for \texttt{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: \url{http://www.cise.ufl.edu/research/sparse/}
symbfact

Purpose
Symbolic factorization analysis

Syntax
- count = symbfact(A)
- count = symbfact(A, 'col')
- count = symbfact(A, 'sym')
- [count, h, parent, post, R] = symbfact(...)

Description
- count = symbfact(A) returns the vector of row counts for the upper
  triangular Cholesky factor of a symmetric matrix whose upper triangle is that
  of A, assuming no cancellation during the factorization. symbfact should be
  much faster than chol(A).
- count = symbfact(A, 'col') analyzes A'*A (without forming it explicitly).
- count = symbfact(A, 'sym') is the same as count = symbfact(A).
- [count, h, parent, post, R] = symbfact(...) has several optional return
  values.
  - h: Height of the elimination tree
  - parent: The elimination tree itself
  - post: Postordering permutation of the elimination tree
  - R: 0-1 matrix whose structure is that of chol(A)

See Also
chol, etree, treelayout
**Purpose**

Symmetric LQ method

**Syntax**

- \( x = \text{symmlq}(A,b) \)
- \( \text{symmlq}(A,b,tol) \)
- \( \text{symmlq}(A,b,tol,maxit) \)
- \( \text{symmlq}(A,b,tol,maxit,M) \)
- \( \text{symmlq}(A,b,tol,maxit,M1,M2) \)
- \( \text{symmlq}(A,b,tol,maxit,M1,M2,x0) \)
- \( \text{symmlq}(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...) \)

**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 \( afun \) such that \( afun(x) \) returns \( A*x \).

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, \( \min(n,20) \).

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

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

symmlq(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...) passes parameters p1,p2,... to functions afun(x,p1,p2,...), m1fun(x,p1,p2,...), and m2fun(x,p1,p2,...).

[x,flag] = symmlq(A,b,tol,maxit,M1,M2,x0,p1,p2,...) 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,tol,maxit,M1,M2,x0,p1,p2,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = symmlq(A,b,tol,maxit,M1,M2,x0,p1,p2,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.
symmlq

\[ \text{x, flag, relres, iter, resvec} = \]
\[ \text{symmlq}(A, b, \text{tol}, \text{maxit}, M1, M2, x0, p1, p2, ...) \]
also returns a vector of estimates of the \text{symmlq} residual norms at each iteration, including \( \|b - Ax0\| \).

\[ \text{x, flag, relres, iter, resvec, resveccg} = \]
\[ \text{symmlq}(A, b, \text{tol}, \text{maxit}, M1, M2, x0, p1, p2, ...) \]
also returns a vector of estimates of the conjugate gradients residual norms at each iteration.

**Examples**

**Example 1.**

\[ n = 100; \]
\[ \text{on} = \text{ones}(n,1); \]
\[ A = \text{spdiags([-2*on 4*on -2*on],[-1:1,n,n]);} \]
\[ b = \text{sum}(A,2); \]
\[ \text{tol} = 1e-10; \]
\[ \text{maxit} = 50; \]
\[ M1 = \text{spdiags(4*on,0,n,n);} \]
\[ x = \text{symmlq}(A, b, \text{tol}, \text{maxit}, M1, [];[],[]); \]
\[ \text{symmlq converged at iteration 49 to a solution with relative residual 4.3e-015} \]

Alternatively, use this matrix-vector product function

\[ \text{function y = afun(x, n)} \]
\[ y = 4 \times x; \]
\[ y(2:n) = y(2:n) - 2 \times x(1:n-1); \]
\[ y(1:n-1) = y(1:n-1) - 2 \times x(2:n); \]

as input to \text{symmlq}.

\[ x1 = \text{symmlq(@afun, b, tol, maxit, M1, [];[],[])}; \]

**Example 2.**

Use a symmetric indefinite matrix that fails with \text{pcg}.

\[ A = \text{diag([20:-1:1,-1:-1:-20])}; \]
\[ b = \text{sum}(A,2); \quad \% \text{The true solution is the vector of all ones.} \]
\[ x = \text{pcg}(A,b); \quad \% \text{Errors out at the first iteration.} \]
\[ \text{pcg stopped at iteration 1 without converging to the desired tolerance 1e-006 because a scalar quantity became too small or too large to continue computing.} \]
symmlq

The iterate returned (number 0) has relative residual 1

However, symmlq can handle the indefinite matrix A.

```matlab
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), / (slash)

References
**Purpose**
Sparse symmetric minimum degree ordering

**Syntax**

\[ p = \text{symmmd}(S) \]

**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.

**Remarks**
The minimum degree ordering is automatically used by \( \backslash \) and \( / \) for the solution of symmetric, positive definite, sparse linear systems.

Some options and parameters associated with heuristics in the algorithm can be changed with \text{spparms}.

**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 \( K' \cdot K \) has the same nonzero structure as \( A \) and then calls the column minimum degree code for \( K \).

**Examples**
Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the \text{symrcm} reference page.

```matlab
B = bucky+4*speye(60);
r = symrcm(B);
p = symmmd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R), title('B(r,r)')
subplot(2,2,2), spy(S), title('B(s,s)')
subplot(2,2,3), spy(chol(R)), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S)), 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, colmmd, colperm, symamd, 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
\[
p = \text{symrcm}(B);
\]
\[
R = B(p,p);
\]
The spy plot shows a much narrower bandwidth.

```matlab
subplot(1,2,2), spy(R), title('B(p,p)')
```

This example is continued in the reference pages for `symmmd`.

The bandwidth can also be computed with

```matlab
[i,j] = find(B);
bw = max(i-j) + 1
```

The bandwidths of `B` and `R` are 35 and 12, respectively.

**See Also**
colamd, colmmd, colperm, symamd, symmmd

**References**


Purpose
Determine the symbolic variables in an 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.

```matlab
symvar 'cos(pi*x - beta1)'
```

```matlab
ans =

'beta1'
'x'
```

See Also
`findstr`
**Purpose**
Run operating system command and return result.

**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.

```
[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, `result` is an empty matrix, and an explanatory message appears.

**Examples**
Display the current directory by accessing the operating system.

```
system('pwd')
```
MATLAB displays the current directory and shows that the command executed correctly because `ans` is 0.

```
D:/mymfiles/
```

```
ans =
0
```
Similarly, run the operating system `pwd` command and assign the current directory to `curr_dir`.

```
[s, curr_dir] = system('pwd')
```
MATLAB displays

```
s =
0
```
```
curr_dir =
D:/mymfiles/
```

**See Also**
`!(exclamation point)`, `dos`, `perl`, `unix`, `winopen`
2tan

Purpose
Tangent of an 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 .

```matlab
x = (-pi/2)+0.01:0.01:(pi/2)-0.01;
plot(x,tan(x)), grid on
```

The expression tan(pi/2) does not evaluate as infinite but as the reciprocal of the floating point accuracy eps since pi is only a floating-point approximation to the exact value of π.

Definition
The tangent can be defined as

```
tan \theta = \frac{\sin \theta}{\cos \theta}
```

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
atan, atan2, tanh
Purpose: Tangent of an 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: atand, tan
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 .

```matlab
x = -5:0.01:5;
plot(x,tanh(x)), grid on
```

Definition
The hyperbolic tangent can be defined as

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
tempdir

Purpose
Return the name of the 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
### Purpose
Unique name for temporary file

### Syntax
\[
\text{tmp\_nam} = \text{tempname}
\]

### Description
\[
\text{tmp\_nam} = \text{tempname}
\]
returns a unique string, \text{tmp\_nam}, suitable for use as a temporary filename.

---

**Note**  The filename that \text{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
\text{tempdir}
**tetramesh**

**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.,

\[ X = [x(:) \ y(:) \ z(:)] \]

**Examples**
Generate a 3-dimensional Delaunay tesselation, then use tetramesh to visualize the tetrahedrons that form the corresponding simplex.

```
d = [-1 1];
```
tetramesh

[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
delaunayn, patch, Patch Properties, trimesh, trisurf
texlabel

**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,

```matlab
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.

```matlab
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" for related functions
Purpose
Create text object in current axes

Syntax
- `text(x,y,'string')`
- `text(x,y,z,'string')`
- `text(...,'PropertyName',PropertyValue...)`
- `h = text(...)`

Description
`text` is the low-level function for creating text graphics objects. Use `text` to place character strings at specified locations.

- `text(x,y,'string')` adds the string in quotes to the location specified by the point `(x,y)`.

- `text(x,y,z,'string')` adds the string in 3-D coordinates.

- `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('PropertyName',PropertyValue...)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

- `h = text(...)` returns a column vector of handles to text objects, one handle per object. All forms of the `text` function optionally return this output argument.

Remarks
Specify the text location coordinates (the `x`, `y`, and `z` arguments) in the data units of the current axes (see “Examples”). 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.

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.

text is a low-level function that accepts property name/property value pairs as input arguments. However, the convenience form,

```plaintext
text(x,y,z,'string')
```

is equivalent to

```plaintext
text('XData',x,'YData',y,'ZData',z,'String','string')
```

You can specify other properties only as property name/property value pairs. See the text property list at the end of this page for a description of each property. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the set and get reference pages for examples of how to specify these data types).

text does not respect the setting of the figure or axes NextPlot property. This allows you to add text objects to an existing axes without setting hold to on.

**Examples**

The statements

```plaintext
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
\[ \text{text}(x,y,'e^{i\omega\tau} = \cos(\omega\tau) + i \sin(\omega\tau)') \]
uses embedded TeX sequences to produce
\[ e^{i\omega\tau} = \cos(\omega\tau) + i \sin(\omega\tau) \]

**See Also**

gtext, int2str, num2str, title, xlabel, ylabel, zlabel

The “Labeling Graphs” topic in the online Using MATLAB Graphics manual discusses positioning text.

See the annotation function for information about text annotations.
Object Hierarchy

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.

Property List
The following table lists all text properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining the Character String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Editing</td>
<td>Enables or disables editing mode</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Interpreter</td>
<td>Enables or disables TeX two levels of interpretation</td>
<td>Values: latex, tex, none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: tex</td>
</tr>
<tr>
<td>String</td>
<td>The character string (including list of TeX character sequences)</td>
<td>Value: character string</td>
</tr>
<tr>
<td>Positioning the character string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extent</td>
<td>Position and size of text object</td>
<td>Values: [left, bottom, width, height]</td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Property Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HorizontalAlignment</td>
<td>Horizontal alignment of text string</td>
<td>Values: left, center, right&lt;br&gt;Default: left</td>
</tr>
<tr>
<td>Position</td>
<td>Position of text Extent rectangle</td>
<td>Values: [x, y, z] coordinates&lt;br&gt;Default: [] (empty matrix)</td>
</tr>
<tr>
<td>Rotation</td>
<td>Orientation of text object</td>
<td>Value: scalar (degrees)&lt;br&gt;Default: 0</td>
</tr>
<tr>
<td>Units</td>
<td>Units for Extent and Position properties</td>
<td>Values: pixels, normalized, inches, centimeters, points, data&lt;br&gt;Default: data</td>
</tr>
<tr>
<td>VerticalAlignment</td>
<td>Vertical alignment of text string</td>
<td>Values: top, cap, middle, baseline, bottom&lt;br&gt;Default: middle</td>
</tr>
</tbody>
</table>

**Text Bounding Box**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackgroundColor</td>
<td>Color of text extent rectangle</td>
<td>Value: ColorSpec&lt;br&gt;Default: none</td>
</tr>
<tr>
<td>EdgeColor</td>
<td>Color of edge drawn around text extent rectangle</td>
<td>Value: ColorSpec&lt;br&gt;Default: none</td>
</tr>
<tr>
<td>LineWidth</td>
<td>Width of the line (in points) used to draw the box around the text extent rectangle</td>
<td>Value: scalar (points)&lt;br&gt;Default: 0.5</td>
</tr>
<tr>
<td>LineStyle</td>
<td>Style of the line used to draw the box around the text extent rectangle</td>
<td>Values: −, −−, ;, −., none&lt;br&gt;Default: −</td>
</tr>
<tr>
<td>Margin</td>
<td>Distance in pixels from the text extent to the edge of the box enclosing the text</td>
<td>Value: scalar (pixels)&lt;br&gt;Default: 2</td>
</tr>
</tbody>
</table>

**Specifying the Font**
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FontAngle</td>
<td>Selects italic-style font</td>
<td>Values: normal, italic, oblique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: normal</td>
</tr>
<tr>
<td>FontName</td>
<td>Selects font family</td>
<td>Value: a font supported by your system or the string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FixedWidth</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: Helvetica</td>
</tr>
<tr>
<td>FontSize</td>
<td>Size of font</td>
<td>Value: size in FontUnits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: 10 points</td>
</tr>
<tr>
<td>FontUnits</td>
<td>Units for FontSize property</td>
<td>Values: points, normalized, inches, centimeters, pixels</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: points</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Weight of text characters</td>
<td>Values: light, normal, demi, bold</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: normal</td>
</tr>
</tbody>
</table>

**Controlling the Appearance**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clipping</td>
<td>Clipping to axes rectangle</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>EraseMode</td>
<td>Method of drawing and erasing the text (useful for animation)</td>
<td>Values: normal, none, xor, background</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: normal</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Highlights text when selected (Selected property is set to on)</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>Visible</td>
<td>Makes the text visible or invisible</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>Color</td>
<td>Color of the text</td>
<td>ColorSpec</td>
</tr>
</tbody>
</table>

**Controlling Access to Text Objects**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandleVisibility</td>
<td>Determines if and when the text’s handle is visible to other functions</td>
<td>Values: on, callback, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Property Value</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>HitTest</td>
<td>Determines if the text can become the current object (see the figure CurrentObject property)</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
</tbody>
</table>
| General Information About Text Objects
| Children       | Text objects have no children.                                                       | Value: [ ] (empty matrix) |
| Parent         | The parent of a text object is an axes hggroup, or hgtransform object.                | Value: object handle    |
| Selected       | Indicates whether the text is in a selected state                                     | Values: on, off         |
|                |                                                                                      | Default: off            |
| Tag            | User-specified label                                                                  | Value: any string       |
|                |                                                                                      | Default: '' (empty string) |
| Type           | The type of graphics object (read only)                                              | Value: the string 'text' |
| UserData       | User-specified data                                                                  | Value: any matrix       |
|                |                                                                                      | Default: [ ] (empty matrix) |
| Controlling Callback Routine Execution
<p>| BeingDeleted   | Query to see if object is being deleted.                                              | Values: on | off |
|                |                                                                                      | Read only              |
| BusyAction     | Specifies how to handle callback routine interruption                               | Values: cancel, queue   |
|                |                                                                                      | Default: queue          |
| ButtonDownFcn  | Defines a callback routine that executes when a mouse button is pressed on over the text | Value: string or function handle |
|                |                                                                                      | Default: '' (empty string) |
| CreateFcn      | Defines a callback routine that executes when a text is created                      | Value: string or function handle |
|                |                                                                                      | Default: '' (empty string) |</p>
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteFcn</td>
<td>Defines a callback routine that executes when the text is deleted (via close or delete)</td>
<td>Value: string or function handle Default: ‘’ (empty string)</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Determines if callback routine can be interrupted</td>
<td>Values: on, off Default: on (can be interrupted)</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Associates a context menu with the text</td>
<td>Value: handle of a uicontextmenu</td>
</tr>
</tbody>
</table>
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. For example, the following code creates a text object that labels a plot and sets the background color to light green.

```matlab
    text(3*pi/4,sin(3*pi/4),...
    ['sin(3*pi/4) = ',num2str(sin(3*pi/4))],...
    'HorizontalAlignment','center',...
    'BackgroundColor',[.7 .9 .7]);
```

![Graph with text annotation](image)

2-2230
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.

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.
**Text Properties**

**ButtonDownFcn** string or function handle

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

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** string or function handle

*Callback routine executed during object creation.* This property defines a callback routine 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

```matlab
set(0,'DefaultTextCreateFcn',...
    'set(gcf,''Pointer'',''crosshair''))
```

defines a default value on the root level that sets the figure `Pointer` property to crosshairs whenever you create a text object. MATLAB executes this routine after setting all text properties. Setting this property on an existing text 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 text callback routine. A callback routine that executes when you delete
the text object (e.g., when you issue a delete command or clear the axes or
figure). 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 the callback function.

EdgeColor

ColorSpec | {none}

Color of edge drawn around text extent rectangle. This property enables you to
specify the color of a box drawn around the text Extent. For example, the
following code draws a red rectangle around text that labels a plot.

text(3*pi/4,sin(3*pi/4),...
'\leftarrow 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)
Text Properties

- **Margin** — Increases the size of the rectangle by adding a margin to the existing text extent rectangle

**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 beginning 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.
**Text Properties**

**Specifying a Fixed-Width Font**

If you want text to use a fixed-width font that looks good in any locale, you should set `FontName` to the string `FixedWidth`:

```matlab
set(text_handle,'FontName','FixedWidth')
```

This eliminates the need to hard-code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan where multibyte character sets are used). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` (note that this string is case sensitive) and rely on `FixedWidthFontName` to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`.

Note that setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

**FontSize**

Size in `FontUnits`

Font size. A value specifying the font size to use for text in units determined by the `FontUnits` property. The default point size is 10 (1 point = 1/72 inch).

**FontWeight**

`light` | `{normal}` | `demi` | `bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `bold` or `demi` causes MATLAB to use a bold font.

**FontUnits**

{`points`} | `normalized` | `inches` | `centimeters` | `pixels`

Font size units. MATLAB uses this property to determine the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

**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

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).

See the Extent property for related information.
Interpretation \( T_{\text{E}}X \) instructions. This property controls whether MATLAB interprets certain characters in the \texttt{String} property as \( T_{\text{E}}X \) instructions (default) or displays all characters literally. See the \texttt{String} property for a list of supported \( T_{\text{E}}X \) instructions.

**Latex Interpreter**

To enable the La\(T\)eX interpreter for text objects, set the \texttt{Interpreter} property to \texttt{latex}. For example, the following statement displays an equation in a figure at the point \([0.5, 0.5]\), and enlarges the font to 16 points.

```
  text('Interpreter','latex',...
       'String','$$\int_0^x\int_y dF(u,v)$$',...
       'Position',[0.5 0.5],...
       'FontSize',16)
```

![Figure 1](image)
Information About Using \TeX

The following references may be useful to people who are not familiar with \TeX.

- The \TeX\ Users Group home page: http://www.tug.org

**Interruptible** \{on\} | off

*Callback routine interruption mode.* The Interruptible property controls whether a text callback routine can be interrupted by subsequently invoked callback routines. Text objects have three properties that define callback routines: ButtonDownFcn, CreateFcn, and DeleteFcn. See the BusyAction property for information on how MATLAB executes callback routines.

**LineStyle** \{-\} | -- | : | -. | none

*Edge line type.* This property determines the line style used to draw the edges of the text Extent. The available line styles are shown in the following table.

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

For example, the following code draws a red rectangle with a dotted line style around text that labels a plot.

```matlab
    text(3*pi/4,sin(3*pi/4),...
         '\leftarrow\sin(t) = .707',...
         'EdgeColor','red',...
         'LineWidth',2,...
         'LineStyle','-.');
```
For additional features, see the following properties:

- **BackgroundColor** — Color of the rectangle’s interior (none by default)
- **EdgeColor** — Color of the rectangle’s edge (none by default)
- **LineWidth** — Width of the rectangle’s edge line (first set EdgeColor)
- **Margin** — Increases the size of the rectangle by adding a margin to the existing text extent rectangle

**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

**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)

**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).
Text Properties

**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>
</tbody>
</table>

2-2244
<table>
<thead>
<tr>
<th>Character Sequence</th>
<th>Symbol</th>
<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>\theta</td>
<td>θ</td>
<td>\Theta</td>
<td>Θ</td>
<td>\leftrightarrow</td>
<td>↔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\vartheta</td>
<td>ϑ</td>
<td>\Lambda</td>
<td>Λ</td>
<td>\leftarrow</td>
<td>←</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\iota</td>
<td>i</td>
<td>\Xi</td>
<td>Ξ</td>
<td>\uparrow</td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\kappa</td>
<td>κ</td>
<td>\Pi</td>
<td>Π</td>
<td>\rightarrow</td>
<td>→</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\lambda</td>
<td>λ</td>
<td>\Sigma</td>
<td>Σ</td>
<td>\downarrow</td>
<td>↓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\mu</td>
<td>μ</td>
<td>\Upsilon</td>
<td>Υ</td>
<td>\circ</td>
<td>°</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\nu</td>
<td>ν</td>
<td>\Phi</td>
<td>Φ</td>
<td>\pm</td>
<td>±</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\xi</td>
<td>ξ</td>
<td>\Psi</td>
<td>Ψ</td>
<td>\geq</td>
<td>≥</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\pi</td>
<td>π</td>
<td>\Omega</td>
<td>Ω</td>
<td>\propto</td>
<td>∝</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\rho</td>
<td>ρ</td>
<td>\forall</td>
<td>∀</td>
<td>\partial</td>
<td>∂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\sigma</td>
<td>σ</td>
<td>\exists</td>
<td>∃</td>
<td>\bullet</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\varsigma</td>
<td>ς</td>
<td>\ni</td>
<td>ι</td>
<td>\div</td>
<td>÷</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\tau</td>
<td>τ</td>
<td>\equiv</td>
<td>≡</td>
<td>\neq</td>
<td>≠</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\approx</td>
<td>≈</td>
<td>\aleph</td>
<td>ℵ</td>
<td>\wp</td>
<td>℘</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\Im</td>
<td>ℑ</td>
<td>\Re</td>
<td>ℜ</td>
<td>\nabla</td>
<td>∇</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\otimes</td>
<td>⊗</td>
<td>\oplus</td>
<td>⊕</td>
<td>\oslash</td>
<td>∅</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\cap</td>
<td>∩</td>
<td>\cup</td>
<td>∪</td>
<td>\supseteq</td>
<td>⊑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\subset</td>
<td>⊂</td>
<td>\subseteq</td>
<td>⊆</td>
<td>\int</td>
<td>∫</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\rfloor</td>
<td>\lfloor</td>
<td>\lceil</td>
<td>\lceil</td>
<td>\backslash{nabla}</td>
<td>∇</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\lfloor</td>
<td>\lfloor</td>
<td>\cdots</td>
<td>\ldots</td>
<td>\ldots</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\perp</td>
<td>⊥</td>
<td>\neg</td>
<td>¬</td>
<td>\prime</td>
<td>′</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You can also specify stream modifiers that control the font used. 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.

Stream modifiers remain in effect until the end of the string or only within the context defined by braces {}.

### Specifying Subscript and Superscript Characters

The subscript character “\_” and the superscript character “\^” modify the character or substring defined in braces immediately following.

To print the special characters used to define the TeX strings when Interpreter is Tex, prefix them with the backslash “\” character: \&, \{|}, \_, \^.

See the example 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.
Text 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. 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.

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.
Text Properties

**VerticalAlignment**  top | cap | {middle} | baseline | bottom

*Vertical alignment of text.* This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the *Position* property. The possible values mean

- **top** — Place the top of the string’s *Extent* rectangle at the specified y-position.
- **cap** — Place the string so that the top of a capital letter is at the specified y-position.
- **middle** — Place the middle of the string at the specified y-position.
- **baseline** — Place font baseline at the specified y-position.
- **bottom** — Place the bottom of the string’s *Extent* rectangle at the specified y-position.

The following picture illustrates the alignment options.

Text *VerticalAlignment* property viewed with the *HorizontalAlignment* property set to *left* (the default).

---

**Visible**  {on} | off

*Text visibility.* By default, all text is visible. When set to off, the text is not visible, but still exists, and you can query and set its properties.
textread

Purpose Read data from text file, write to multiple outputs

Graphical Interface As an alternative to textread, use the Import Wizard. To activate the Import Wizard, select Import Data from the File menu.

Syntax

\[
[A,B,C,...] = \text{textread}('filename', 'format')
\]

\[
[A,B,C,...] = \text{textread}('filename', 'format', N)
\]

\[
[...] = \text{textread}(..., 'param', 'value', ...)
\]

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. textread is useful for reading text files with a known format. textread handles both fixed and free format files.

Note When reading large text files, reading from a specific point in a file, or reading file data into a cell array rather than multiple outputs, you might prefer to use the textscan function.

textread matches and converts groups of characters from the input. Each input field is defined as a string of non-white-space characters that extends to the next white-space or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated white-space characters are treated as one.

The format string determines the number and types of return arguments. The number of return arguments is the number of items in the format string. The format string supports a subset of the conversion specifiers and conventions of the C language fscanf routine. Values for the format string are listed in the table below. White-space characters in the format string are ignored.
<table>
<thead>
<tr>
<th>format</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literals (ordinary characters)</td>
<td>Ignore the matching characters. For example, in a file that has Dept followed by a number (for department number), to skip the Dept and read only the number, use 'Dept' in the format string.</td>
<td>None</td>
</tr>
<tr>
<td>%d</td>
<td>Read a signed integer value.</td>
<td>Double array</td>
</tr>
<tr>
<td>%u</td>
<td>Read an integer value.</td>
<td>Double array</td>
</tr>
<tr>
<td>%f</td>
<td>Read a floating-point value.</td>
<td>Double array</td>
</tr>
<tr>
<td>%s</td>
<td>Read a white-space or delimiter-separated string.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>%q</td>
<td>Read a string, which could be in double quotes.</td>
<td>Cell array of strings. Does not include the double quotes.</td>
</tr>
<tr>
<td>%c</td>
<td>Read characters, including white space.</td>
<td>Character array</td>
</tr>
<tr>
<td>[%...</td>
<td>Read the longest string containing characters specified in the brackets.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>[%^...</td>
<td>Read the longest nonempty string containing characters that are not specified in the brackets.</td>
<td>Cell array of strings</td>
</tr>
<tr>
<td>%*...instead of %</td>
<td>Ignore the matching characters specified by *.</td>
<td>No output</td>
</tr>
<tr>
<td>%w...instead of %</td>
<td>Read field width specified by w. The %f format supports %w.pf, where w is the field width and p is the precision.</td>
<td>No output</td>
</tr>
</tbody>
</table>
textread

[A,B,C,...] = textread('filename','format',N) reads the data, reusing the format string N times, where N is an integer greater than zero. If N is smaller than zero, textread reads the entire file.

[...] = textread(...,'param','value',...) customizes textread using param/value pairs, as listed in the table below.

<table>
<thead>
<tr>
<th>param</th>
<th>value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Space</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>Backspace</td>
<td>Backspace</td>
</tr>
<tr>
<td></td>
<td>Newline</td>
<td>Newline</td>
</tr>
<tr>
<td></td>
<td>Carriage return</td>
<td>Carriage return</td>
</tr>
<tr>
<td></td>
<td>Horizontal tab</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></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>
<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 '\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>
</tbody>
</table>
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.

### 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.

    [names, types, y, answer] = textread('mydata.dat', ...    
      '%9c %5s %*f %2d %3s', 1)

returns

    names =    
      Sally     
    types =     
      'Level1'    
    y =     
      45     
    answer =     
      'Yes'

%*f in the format string causes textread to ignore the floating point value, in this case, 12.34.

Example 3 — Read Using Literal to Ignore Matching Characters

The first line of mydata.dat is

    Sally    Level1 12.34 45 Yes

Read the first line of the file, ignoring the characters Type in the second field.

    [names, typenum, x, y, answer] = textread('mydata.dat', ...    
      '%s Type%d %f %d %s', 1)

returns

    names =    
      'Sally'    
    typenum =    
      1    
    x =    
      12.34000000000000
\texttt{y = 45} \\
\texttt{answer = 'Yes'}

Type\%d in the format string causes the characters \texttt{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.

\textbf{Example 4 — Specify Value to Fill Empty Cells}
For files with empty cells, use the emptyvalue parameter. Suppose the file \texttt{data.csv} contains:

\begin{verbatim}
1,2,3,4,,6 \\
7,8,9,,11,12
\end{verbatim}

Read the file using NaN to fill any empty cells:

\begin{verbatim}
data = textread('data.csv', '', 'delimiter', ',', ... 'emptyvalue', NaN);
\end{verbatim}

\textbf{Example 5 — Read M-File into a Cell Array of Strings}
Read the file \texttt{fft.m} into cell array of strings.

\begin{verbatim}
file = textread('fft.m', '%s', 'delimiter', '\n', 'whitespace', '');
\end{verbatim}

\textbf{See Also} \texttt{textscan}, \texttt{dlmread}, \texttt{csvread}, \texttt{fscanf}
Purpose
Read data from text file, convert, and write to cell array

Syntax
C = textscan(fid, 'format')
C = textscan(fid, 'format', N)
C = textscan(fid, 'format', param, value, ...)
C = textscan(fid, 'format', N, param, value, ...)

Description
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 the format string. 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. If N is -1, or is unspecified, textscan reads the entire file. 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 2-2264 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.

Note
If textscan fails to convert a data field, it stops reading and returns all fields read before the failure. You can resume reading from the same file by calling textscan again using the same file identifier, fid.
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 seek to any position in the file and begin the textscan at that point. The textread function requires that you start reading from the beginning of the file.
- Subsequent textscans start reading the file at the point where the last textscan left off. The textread function always begins at the start of the file, regardless of any prior textread.
- 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 regards a text file as consisting 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 2-2264). 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 set end-of-line to a different character sequence by specifying an ‘endofline’ parameter in the textscan command (see “User Configurable Options” on page 2-2264). If you set the delimiter parameter to ‘EOL’ (using the third syntax shown above), textscan reads complete rows.
# 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 double.</td>
</tr>
<tr>
<td>%d</td>
<td>Read a number and convert to int32.</td>
</tr>
<tr>
<td>%d8</td>
<td>Read a number and convert to int8.</td>
</tr>
<tr>
<td>%d16</td>
<td>Read a number and convert to int16.</td>
</tr>
<tr>
<td>%d32</td>
<td>Read a number and convert to int32.</td>
</tr>
<tr>
<td>%d64</td>
<td>Read a number and convert to int64.</td>
</tr>
<tr>
<td>%u</td>
<td>Read a number and convert to uint32.</td>
</tr>
<tr>
<td>%u8</td>
<td>Read a number and convert to int8.</td>
</tr>
<tr>
<td>%u16</td>
<td>Read a number and convert to int16.</td>
</tr>
<tr>
<td>%u32</td>
<td>Read a number and convert to int32.</td>
</tr>
<tr>
<td>%u64</td>
<td>Read a number and convert to int64.</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>
</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:

```matlab
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:

```matlab
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

```matlab
C{:}
an =
```
'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 then reread the line, this time skipping the second, fifth, and sixth fields:

```matlab
fseek(fid, 0, -1);
C = textscan(fid, '%s %*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:

```matlab
str = '';
for k = 1:length(C)
    str = [str char(C{k}) ' '];
    if k == 4, disp(str), end
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,

```
Sally    Level1  12.34
Joe      Level2  23.54
Bill     Level3  34.90
```

this command removes the substring 'Level' from the output and converts the level number to a `uint8`:

```matlab
C = textscan(fid, '%s Level%u %f');
```
This returns a cell array `C` with the second cell containing only the unsigned integers:

```matlab
C{1} = {'Sally'; 'Joe'; 'Bill'}       class cell
C{2} = [1; 2; 3]                      class uint8
C{3} = [12.34; 23.54; 34.90]          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 2-2258. 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 `%9.1f` 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><code>%n</code>, <code>%d</code>, <code>%u</code>, <code>%f</code>, and variants thereof</td>
<td>Read to the first delimiter. Example: <code>%n</code> reads <code>'473.238 '</code> as <code>473.238</code>.</td>
</tr>
<tr>
<td><code>%Nn</code>, <code>%Nd</code>, <code>%Nu</code>, <code>%Nf</code>, and variants thereof</td>
<td>Read <code>N</code> digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Example: <code>%5f32</code> reads <code>'473.238 '</code> as <code>473.2</code>.</td>
</tr>
<tr>
<td>Specifiers that start with <code>%N.Df</code></td>
<td>Read <code>N</code> digits (counting a decimal point as a digit), or up to the first delimiter, whichever comes first. Return <code>D</code> decimal digits in the output. Example: <code>%7.2f</code> reads <code>'473.238 '</code> as <code>473.23</code>.</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.

<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 up to the first character not specified within the brackets (i.e., read up to the first character that is not an a, b, or c).  
                   Example: %[mus] reads 'summer' as 'summ'. |
| %N[abc]          | Read N characters, or up to the first character not specified within the brackets, whichever comes first.  
                   Example: %2[mus] reads 'summer' as 'su'. |
| %[^abc]          | Read up to the first character that is specified within the brackets, (i.e., read up to the first occurrence of an a, b, or c).  
                   Example: %[^xrg] reads 'summer' as 'summe'. |
| %N[^abc]         | Read N characters, or up to the first character that is specified within the brackets, whichever comes first.  
                   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>
<tbody>
<tr>
<td>%c</td>
<td>Read one character. Example: %c reads 'Let's go!' as 'L'.</td>
</tr>
<tr>
<td>%Nc</td>
<td>Read N characters, including delimiter characters. Example: %9c reads 'Let's go!' as 'Let's go!'.</td>
</tr>
</tbody>
</table>

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 examining nonnumeric fields for custom empty values. See “User Configurable Options” on page 2-2264.

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.

<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>commentStyle</td>
<td>Symbol(s) designating text to be ignored (see “Values for commentStyle”, below)</td>
<td>None</td>
</tr>
<tr>
<td>delimiter</td>
<td>Delimiter characters</td>
<td>None</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>returnOnError</td>
<td>Behavior on failing to read or convert (1=true or 0)</td>
<td>1</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>
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>

Examples

Example 1 — Reading Different Types of Data

Text file scan1.dat contains data in the following form:

<table>
<thead>
<tr>
<th>Name</th>
<th>Level</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
<th>Value 6</th>
<th>Value 7</th>
<th>Value 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sally</td>
<td>Level1</td>
<td>12.34</td>
<td>45</td>
<td>1.23e10</td>
<td>Inf</td>
<td>NaN</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joe</td>
<td>Level2</td>
<td>23.54</td>
<td>60</td>
<td>9e19</td>
<td>-inf</td>
<td>0.001</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bill</td>
<td>Level3</td>
<td>34.90</td>
<td>12</td>
<td>2e5</td>
<td>10</td>
<td>100</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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; 34.90]  class single
- C{4} = [45; 60; 12]  class int8
- C{5} = [1.23e10; 9e19; 2e5]  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 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
- \( C\{4\} = [1.23e10; 9e19; 2e5] \) class uint32
- \( C\{5\} = [\text{Inf}; -\text{Inf}; 10] \) class double
- \( C\{6\} = [\text{NaN}; 0.001; 100] \) 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%*[\n\']);
fclose(fid);
```

textscan returns a 1-by-1 cell array \( \text{names} \):

```matlab
size(names)
an =
1
1
```

The one cell contains

- \( \text{names}{1} = \{'Sally'; 'Joe'; 'Bill'\} \) 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:

```matlab
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} = [1.23e10; 9e19; 2e5]           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:

```matlab
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:

```matlab
file =
{37x1 cell}
```

Show the first three lines of the file:

```matlab
lines = file{1};
lines{1:3, :}
ans =
't function [varargout] = fft(varargin)'
ans =
'FFT Discrete Fourier transform.'
ans =
' % FFT(X) is the discrete Fourier transform (DFT) of vector X. For'
```

**Example 6 — Using a Nondefault Empty Value**

Read files with empty cells, setting the `emptyvalue` parameter. The file `data.csv` contains
Read the file as shown here, using -Inf in empty cells:

```matlab
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
- \( C\{4\} = [4; \text{NaN}] \) class double
- \( C\{5\} = [-\text{Inf}; 11] \) 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
```

See Also
dlmread, dlmwrite, xlswrite, fopen, importdata
textwrap

Purpose
Return 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  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.

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
**timer**

**Purpose**
Construct timer object

**Syntax**

```matlab
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 2-2272 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, disp, get, isvalid, set, 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` function. To set the value of the properties of a timer object, use the `set` function.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Data Types, Values, and Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>AveragePeriod</td>
<td>The average time between <code>TimerFcn</code> executions since the timer started. Note: Value is NaN until timer executes two timer callbacks.</td>
<td>Data type: double, Default: NaN, Read only: Always</td>
</tr>
</tbody>
</table>
| BusyMode        | Action taken when a timer has to execute `TimerFcn` before the completion of previous execution of `TimerFcn`.  
                  - 'drop' — Do not execute the function.  
                  - 'error' — Generate an error.  
                  - 'queue' — Execute function at next opportunity. | Data type: Enumerated string, Values: 'drop', 'error', 'queue', Default: 'drop', Read only: While Running = 'on' |
<p>| ErrorFcn        | Function that the timer executes when an error occurs. This function executes before the <code>StopFcn</code>. See “Creating Timer Callback Functions” for more information. | Data type: Text string, function handle, or cell array, Default: None, Read only: Never          |
| ExecutionMode   | Determines how the timer object schedules timer events. See “Timer Execution Modes” for more information. | Data type: Enumerated string, Values: 'singleShot', 'fixedDelay', 'fixedRate', 'fixedSpacing', Default: 'singleShot', Read only: While Running = 'on' |</p>
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Data Types, Values, and Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstantPeriod</td>
<td>The time between the last two executions of TimerFcn.</td>
<td>Data type: double&lt;br&gt;Default: NaN&lt;br&gt;Read only: Always</td>
</tr>
<tr>
<td>Name</td>
<td>User-supplied name</td>
<td>Data type: Text string&lt;br&gt;Default: 'timer-i', where i is a number indicating the i\textsuperscript{th} timer object created this session. To reset i to 1, execute the clear classes command. Never&lt;br&gt;Read only:</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 timer-find 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&lt;br&gt;Values: 'off', 'on'&lt;br&gt;Default: 'on'&lt;br&gt;Read only: Never</td>
</tr>
<tr>
<td>Period</td>
<td>Specifies the delay, in seconds, between executions of TimerFcn.</td>
<td>Data type: double&lt;br&gt;Value: Any number &gt; 0.001&lt;br&gt;Default: 1.0&lt;br&gt;While Running = 'on'</td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Data Types, Values, and Defaults</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Running</td>
<td>Indicates whether the timer is currently executing.</td>
<td>Data type: Enumerated string  'off'  'on'  Default: 'off'  Read only: Always</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  Value: Any number &gt;= 0  Default: 0  Read only: While Running = 'on'</td>
</tr>
<tr>
<td>StartFcn</td>
<td>Function the timer calls when it starts. See “Creating Timer Callback Functions” for more information.</td>
<td>Data type: Text string, function handle, or cell array  Default: None  Read only: Never</td>
</tr>
<tr>
<td>StopFcn</td>
<td>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 Timer Callback Functions” for more information.</td>
<td>Data type: Text string, function handle, or cell array  Default: None  Read only: Never</td>
</tr>
<tr>
<td>Tag</td>
<td>User supplied label</td>
<td>Data type: Text string  Default: '' (empty string)  Read only: Never</td>
</tr>
</tbody>
</table>
## timer

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Data Types, Values, and Defaults</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; Value: Any number &gt; 0; Default: 1; Read only: Never</td>
</tr>
<tr>
<td>TasksExecuted</td>
<td>The number of times the timer has executed TimerFcn since the timer was started.</td>
<td>Data type: double; Value: Any number &gt;= 0; Default: 0; Read only: Always</td>
</tr>
<tr>
<td>TimerFcn</td>
<td>Timer callback function. See “Creating Timer Callback Functions” for more information.</td>
<td>Data type: Text string, function handle, or cell array; Default: None; Read only: Never</td>
</tr>
<tr>
<td>Type</td>
<td>Identifies the object type</td>
<td>Data type: Text string; Value: 'timer'; Read only: 'Always'</td>
</tr>
<tr>
<td>UserData</td>
<td>User-supplied data</td>
<td>Data type: User-defined; Default: []; 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'.

Examples

These examples use timerfind to find timer objects with the specified property values.
timerfind

```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, timerfindall
**Purpose**  
Find timer objects, including invisible objects

**Syntax**

\[
\text{out} = \text{timerfindall} \\
\text{out} = \text{timerfindall}(\text{'P1', V1, 'P2', V2,...}) \\
\text{out} = \text{timerfindall}(S) \\
\text{out} = \text{timerfindall}(\text{obj, 'P1', V1, 'P2', V2,...})
\]

**Description**

\[
\text{out} = \text{timerfindall} \text{returns an array, out, containing all the timer objects} \\
\text{that exist in memory, regardless of the value of the object's ObjectVisibility property.}
\]

\[
\text{out} = \text{timerfindall(}\text{'P1', V1, 'P2', V2,...}) \text{returns an array, out, of} \\
\text{timer objects whose property values match those passed as parameter/value pairs, P1, V1, P2, V2. Parameter/value pairs may be specified as a cell array.}
\]

\[
\text{out} = \text{timerfindall}(S) \text{returns an array, out, of timer objects whose property values match those defined in the structure, S. The field names of S are timer object property names and the field values are the corresponding property values.}
\]

\[
\text{out} = \text{timerfindall}(\text{obj, 'P1', V1, 'P2', V2,...}) \text{restricts the search for matching parameter/value pairs to the timer objects listed in obj. obj can be an array of timer objects.}
\]

**Note**  
When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `timerfindall`.

Note that, for most properties, `timerfindall` performs case-sensitive searches of property values. For example, if the value of an object's Name property is 'MyObject', `timerfindall` will not find a match if you specify 'myobject'. Use the get function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, `timerfindall` will find an object with an ExecutionMode property value of 'singleShot' or 'singleshot'.

**Examples**

Create several timer objects.
timerfindall

```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`, `timerfind`
Purpose

Add title to current axes

Syntax

```
title('string')
title(fname)
title(...,'PropertyName',PropertyValue,...)
```

```
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.
```

```
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” for related functions

Adding Titles to Graphs for more information on ways to add titles
Purpose
Convert CDF epoch object to MATLAB datenum

Syntax
n = todatenum(ep_obj)

Description
n = todatenum(obj) converts the CDF epoch object ep_obj into a MATLAB serial date number. Note that a CDF epoch is the number of milliseconds since 01-Jan-0000 whereas a MATLAB datenum is the number of days since 00-Jan-0000.

Examples
Construct a CDF epoch object from a date string, and then convert the object back into a MATLAB date string:

```matlab
dstr = datestr(today)
dstr =
  08-Oct-2003

obj = cdfepoch(dstr)
obj =
  cdfepoch object:
  08-Oct-2003 00:00:00

dstr2 = datestr(todatenum(obj))
dstr2 =
  08-Oct-2003
```

See Also
cdfepoch, cdfinfo, cdfread, cdfwrite, datenum
Purpose
Toeplitz matrix

Syntax
T = toeplitz(c,r)
T = toeplitz(r)

Description
A Toeplitz matrix is defined by one row and one column. A symmetric Toeplitz matrix is defined by just one row. toeplitz generates Toeplitz matrices given just the row or row and column description.

T = toeplitz(c,r) returns a nonsymmetric Toeplitz matrix T having c as its first column and r as its first row. If the first elements of c and r are different, a message is printed and the column element is used.

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

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
**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.  
\[ t = \text{sum}(	ext{diag}(A)); \]

**See Also**  
det, eig
Purpose
Trapezoidal numerical integration

Syntax
Z = trapz(Y)
Z = trapz(X,Y)
Z = trapz(...,dim)

Description
Z = trapz(Y) computes an approximation of the integral of Y via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply Z by the spacing increment.

If Y is a vector, trapz(Y) is the integral of Y.

If Y is a matrix, trapz(Y) is a row vector with the integral over each column.

If Y is a multidimensional array, trapz(Y) works across the first nonsingleton dimension.

Z = trapz(X,Y) computes the integral of Y with respect to X using trapezoidal integration.

If X is a column vector and Y an array whose first nonsingleton dimension is length(X), trapz(X,Y) operates across this dimension.

Z = trapz(...,dim) integrates across the dimension of Y specified by scalar dim. The length of X, if given, must be the same as size(Y,dim).

Examples
The exact value of $\sin$ is 2.

To approximate this numerically on a uniformly spaced grid, use

\[
\begin{align*}
  X &= 0:pi/100:pi; \\
  Y &= \sin(x);
\end{align*}
\]

Then both

\[
Z = \text{trapz}(X,Y)
\]

and

\[
Z = \pi/100*\text{trapz}(Y)
\]

produce
\[ Z = 1.9998 \]

A nonuniformly spaced example is generated by

\[
\begin{align*}
X &= \text{sort} (\text{rand}(1,101) \times \pi); \\
Y &= \sin(X); \\
Z &= \text{trapz}(X,Y);
\end{align*}
\]

The result is not as accurate as the uniformly spaced grid. One random sample produced

\[
\begin{align*}
Z &= 1.9984
\end{align*}
\]

**See Also**

cumsum, cumtrapz
**treelayout**

**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.

### See Also
- `etree`, `etreeplot`, `treelayout`
Purpose
Lower triangular part of a matrix

Syntax
L = tril(X)
L = tril(X,k)

Description
L = tril(X) returns the lower triangular part of X.

L = tril(X,k) returns the elements on and below the kth diagonal of X. k = 0
is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the
main diagonal.

Examples
tril(ones(4,4),-1)

ans =

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

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 Tri as a mesh. Each row of 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 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 = trimesh(...) returns a handle to a patch graphics object.

Example

Create vertex vectors and a face matrix, then create a triangular mesh plot.

\[
x = \text{rand}(1,50);
y = \text{rand}(1,50);
z = \text{peaks}(6\times 3,6\times 3);
tri = \text{delaunay}(x,y);
\text{trimesh}(\text{tri},x,y,z)
\]

See Also

patch, tetramesh, triplot, trisurf, delaunay

“Creating Surfaces and Meshes” for related functions
triplequad

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 fun(x,y,z) over the three dimensional rectangular region xmin <= x <= xmax, ymin <= y <= ymax, zmin <= z <= zmax. fun is a function handle for either an M-file function or an anonymous function. 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 online MATLAB documentation, explains how to provide addition parameters to the function fun, if necessary.

triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol) uses a tolerance tol instead of the default, which is 1.0e-6.

triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method) uses the quadrature function specified as method, instead of the default quad. Valid values for method are @quadl or the function handle of a user-defined quadrature method that has the same calling sequence as quad and quadl.

Examples
Pass M-file function handle @integrnd to triplequad:

Q = triplequad(@integrnd,0,pi,0,1,-1,1);

where the M-file integrnd.m is

function f = integrnd(x,y,z)
f = y*sin(x)+z*cos(x);

Pass anonymous function handle F to triplequad:

F = @(x,y,z)y*sin(x)+z*cos(x);
Q = triplequad(F,0,pi,0,1,-1,1);

This example integrates y*sin(x)+z*cos(x) over the region 0 <= x <= pi, 0 <= y <= 1, -1 <= z <= 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.

    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 \( \text{Tri} \) as a surface. 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.

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.

\( h = \text{trisurf}(...) \) returns a patch handle.

**Example**

Create vertex vectors and a face matrix, then create a triangular surface plot.

\[
\begin{align*}
x &= \text{rand}(1,50); \\
y &= \text{rand}(1,50); \\
z &= \text{peaks}(6*x 3, 6*x 3); \\
\text{tri} &= \text{delaunay}(x,y); \\
\text{trisurf}(\text{tri},x,y,z)
\end{align*}
\]

**See Also**

patch, surf, tetramesh, trimesh, triplot, delaunay

“Creating Surfaces and Meshes” for related functions
triu

Purpose
Upper triangular part of a matrix

Syntax
U = triu(X)
U = triu(X,k)

Description
U = triu(X) returns the upper triangular part of X.

U = triu(X,k) returns the element on and above the kth diagonal of X. k = 0 is the main diagonal, k > 0 is above the main diagonal, and k < 0 is below the main diagonal.

Examples
triu(ones(4,4),-1)

ans =

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

See Also
diag, tril
**Purpose**
True array

**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.

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**

Begin try block

**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 lasterr, 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 lasterr.

**See Also**

catch, end, lasterr, eval, evalin
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

delaunay, delaunayn, dsearch, tsearchn
**tsearchn**

**Purpose**
N-dimensional closest simplex search

**Syntax**

```matlab
 t = tsearchn(X,TES,XI)
 [t,P] = tsearchn(X,TES,XI)
```

**Description**

`t = tsearchn(X,TES,XI)` returns the indices `t` of the enclosing simplex of the Delaunay tessellation `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. `tsearchn` returns `NaN` for all points outside the convex hull of `X`. `tsearchn` requires a tessellation `TES` of the points `X` obtained from `delaunayn`.

`[t,P] = tsearchn(X,TES,XI)` also returns the barycentric coordinate `P` of `XI` in the simplex `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.

**See Also**
`delaunayn`, `griddatan`, `tsearch`
Purpose
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.

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.

Note Include code for uicontrols of style 'radiobutton' and 'togglebutton' that are managed by a uibuttongroup in the SelectionChangeFcn callback function, not in the individual uicontrol Callback functions. uibuttongroup overwrites the Callback properties of radio buttons and toggle buttons that it manages. See the SelectionChangeFcn property and the example on this reference page for more information.

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.

This table lists all properties useful to uibuttongroup objects, grouping them by function. Each property name acts as a link to a description of the property. Curly braces denote the default value, if any.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controlling Style and Appearance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BackgroundColor</td>
<td>Color of the uibuttongroup background</td>
<td>ColorSpec. Default is the same as the default uicontrol background.</td>
</tr>
<tr>
<td>BorderType</td>
<td>Type of border around the uibuttongroup area.</td>
<td>[none</td>
</tr>
<tr>
<td>BorderWidth</td>
<td>Width in pixels of the button group border.</td>
<td>Integer. Default is 1.</td>
</tr>
<tr>
<td>Clipping</td>
<td>Clipping of child axes, uipanels, and uibuttongroups to the uibuttongroup. Does not affect child uicontrols.</td>
<td>[{on}</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Title font color and color of 2-D border line</td>
<td>ColorSpec. Default is [0 0 0] (black).</td>
</tr>
<tr>
<td>HighlightColor</td>
<td>3-D frame highlight color</td>
<td>ColorSpec. Default is [1 1 1] (white).</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Object highlighted when selected</td>
<td>[{on}</td>
</tr>
<tr>
<td>ShadowColor</td>
<td>3-D frame shadow color</td>
<td>ColorSpec. Default is [.5 .5 .5] (grey).</td>
</tr>
</tbody>
</table>
Visible | Uibuttongroup visibility. Note: Controls the Visible property of child axes, uipanels, anduibuttongroups. Does not affect child uicontrols. |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[{on}]{off}</td>
<td></td>
</tr>
</tbody>
</table>

**General Information About the Object**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Children</td>
<td>All children of the uibuttongroup object</td>
<td>Vector of handles</td>
</tr>
<tr>
<td>Parent</td>
<td>uibuttongroup object's parent</td>
<td>Scalar figure, uipanel, oruibuttongroup handle</td>
</tr>
<tr>
<td>Selected</td>
<td>Whether object is selected</td>
<td>{on}{off}</td>
</tr>
<tr>
<td>SelectedObject</td>
<td>Currently selected radio button or toggle button</td>
<td>Scalar handle. Default is the first uicontrol radio button or toggle button added. Set to [] for no selection.</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
<td>String</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
<td>Matrix</td>
</tr>
</tbody>
</table>

**Controlling the Object Position**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>Button group position relative to parent figure, panel, or button group</td>
<td>Position spec [x y w h]. Default is [0 0 1 1]</td>
</tr>
<tr>
<td>Units</td>
<td>Units used to interpret the position vector</td>
<td>[inches</td>
</tr>
</tbody>
</table>
### Controlling Fonts and Labels

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FontAngle</td>
<td>Title font angle</td>
<td>{normal}</td>
</tr>
<tr>
<td>FontName</td>
<td>Title font name</td>
<td>String. Default is system dependent.</td>
</tr>
<tr>
<td>FontSize</td>
<td>Title font size</td>
<td>Integer. Default is system dependent.</td>
</tr>
<tr>
<td>FontUnits</td>
<td>Title font units</td>
<td>{inches}</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Title font weight</td>
<td>{light}</td>
</tr>
<tr>
<td>Title</td>
<td>Title string</td>
<td>String</td>
</tr>
<tr>
<td>TitlePosition</td>
<td>Location of title string in relation to the button group</td>
<td>{lefttop}</td>
</tr>
</tbody>
</table>

### Controlling Callback Routine Execution

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyAction</td>
<td>Interruption of other callback routines</td>
<td>{queue}</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Button-press callback routine</td>
<td>String or function handle</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
<td>String or function handle</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
<td>String or function handle</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
<td>{on}</td>
</tr>
</tbody>
</table>
This example creates a `uibuttongroup` with three radiobuttons. It manages the radiobuttons with the `SelectionChangeFcn` callback, `selcbk`.

When you select a new radio button, `selcbk` displays the `uibuttongroup` handle on one line, the `EventName`, `OldValue`, and `NewValue` fields of the event data structure on a second line, and the value of the `SelectedObject` property on a third line.

```matlab
h = uibuttongroup('visible','off','Position',[0 0 .2 1]);
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');
set(h,'SelectionChangeFcn',@selcbk);
set(h,'SelectedObject',[]);  % No selection
set(h,'Visible','on');
```

### Examples

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResizeFcn</td>
<td>User-specified resize routine</td>
<td>String or function handle</td>
</tr>
<tr>
<td>SelectionChangeFcn</td>
<td>Callback routine executed when the selected radio button or toggle button changes.</td>
<td>String or function handle</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Associates a uicontextmenu with the uibuttongroup</td>
<td>Scalar handle</td>
</tr>
</tbody>
</table>

**Controlling Access to Objects**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandleVisibility</td>
<td>Handle accessibility from commandline and GUIs</td>
<td>{on}</td>
</tr>
<tr>
<td>HitTest</td>
<td>Selectable by mouse click</td>
<td>{on}</td>
</tr>
</tbody>
</table>
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
Uibuttongroup 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 uibuttongroup background</td>
</tr>
<tr>
<td>BorderType</td>
<td>Type of border around the uibuttongroup area.</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 uibuttongroup</td>
</tr>
<tr>
<td>Clipping</td>
<td>Clipping of child axes, uipanels, and uibuttongroups to the uibuttongroup. 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>
</tbody>
</table>
# Uibuttongroup Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>HitTest</td>
<td>Selectable by mouse click</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>Uibuttongroup position relative to parent figure, uipanel, or uibuttongroup</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>
<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>UICustomMenu</td>
<td>Associates uicontext menu with the uibuttongroup</td>
</tr>
</tbody>
</table>
Uibuttongroup Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>Uibuttongroup visibility. Note: Controls the Visible property of child axes, uipanels, and uibuttongroups. Does not affect child uicontrols.</td>
</tr>
</tbody>
</table>

 setBackgroundColor  ColorSpec

Color of the uibuttongroup background. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the ColorSpec reference page for more information on specifying color.

BorderType  none | {etchedin} | etchedout | beveledin | beveledout | line

Border of the uibuttongroup area. Used to define the button group area graphically. Etched and beveled borders provide a 3-D look. 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.
**Uibuttongroup Properties**

**Note** If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the `Interruptible` property for information about controlling a callback's interruptibility.

**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.
Uibuttongroup Properties

Setting this property on an existing uibuttongroup object has no effect.

To define a default CreateFcn callback for all new uibuttongroups you must define the same default for all uipanels. This default applies unless you override it by specifying a different CreateFcn callback when you call uibuttongroup. For example, the code

```matlab
set(0,'DefaultUipanelCreateFcn','set(gcbo,...
    ''FontName'',''arial'',''FontSize'',12))
```

creates a default CreateFcn callback that runs whenever you create a new panel or button group. It sets the default font name and font size of the uipanel or uibuttongroup title.

To override this default and create a button group whose FontName and FontSize properties are set to different values, call uibuttongroup with code similar to

```matlab
hpt = uibuttongroup(...,'CreateFcn','set(gcbo,...
    ''FontName'',''times'',''FontSize'',14))
```

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uibuttongroup call. In the example above, if instead of redefining the CreateFcn property for this uibuttongroup, you had explicitly set FontSize to 14, the default CreateFcn callback would have set FontSize back to the system dependent default.

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.
UIColor

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.
**Uibuttongroup 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.

**HitTest**  {on} | off

Selectable by mouse click. HitTest determines if the figure 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 figure. If HitTest is off, clicking the figure sets the CurrentObject to the empty matrix.

**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 are 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.

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.
**Uibuttongroup Properties**

**Selected**  
on | off (read only)

*Is object selected?* This property indicates whether the figure 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.

**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 is the handle of the `uibuttongroup`. The second argument is an event data structure that contains the fields shown in the following table.

<table>
<thead>
<tr>
<th>Event Data Structure Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventName</td>
<td>‘SelectionChanged’</td>
</tr>
<tr>
<td>OldValue</td>
<td>Handle of the object selected before this event. <code>[]</code> if none was selected.</td>
</tr>
<tr>
<td>NewValue</td>
<td>Handle of the currently selected object.</td>
</tr>
</tbody>
</table>

Include code for `uicontrol` radio buttons and toggle buttons that are managed by a `uibuttongroup` in the `SelectionChangeFcn` callback function, not in the individual `uicontrol` `Callback` functions. `uibuttongroup` overwrites the Callback properties of radio buttons and toggle buttons that it manages.
SelectionHighlight  \{on\}  |  off

Object highlighted when selected. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

ShadowColor  \{ColorSpec\}

3-D frame shadow color. ShadowColor is a three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the ColorSpec reference page for more information on specifying color.

Tag  \{string\}

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.

With the findobj function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the Tag value 'FormatTb'.

   h = findobj(figurehandles,'Tag','FormatTb')

Title  \{string\}

Title string. The text displayed in the button group title. You can position the title using the TitlePosition property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string 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 uibuttongroup title.

Setting a property value to default, remove, or factory produces the effect described in Setting Default Values. To set Title to one of these words, you must precede the word with the backslash character. For example,

   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.
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. All units are measured from the lower-left corner of the figure window.

- Normalized units map the lower-left corner of the figure window to (0,0) and the upper-right corner to (1.0,1.0).
- pixels, inches, centimeters, and points are absolute units (1 point = 1/72 inch).
- Character units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of Units, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume Units is set to the default value.

UserData matrix

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

Visible {on} | off

Uibuttongroup visibility. By default, a uibuttongroup object is visible. When set to off, the uibuttongroup is not visible, 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 a context menu

**Syntax**
handle = uicontextmenu('PropertyName',PropertyValue,...);

**Description**
uicontextmenu creates a context menu, which is a menu that appears when the user right-clicks on a graphics object.
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.

**Properties**
This table lists all properties useful to uibuttongroup objects, grouping them by function. Each property name acts as a link to a description of the property. Curly braces denote the default value, if any.

This table lists the properties that are useful to uicontextmenu objects, grouping them by function. Each property name acts as a link to a description of the property.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controlling Style and Appearance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td>Uicontextmenu visibility</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Position</td>
<td>Location of uicontextmenu when Visible is set to on</td>
<td>Value: two-element vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: [0 0]</td>
</tr>
<tr>
<td><strong>General Information About the Object</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children</td>
<td>The uimenus defined for the uicontextmenu</td>
<td>Value: matrix</td>
</tr>
<tr>
<td>Parent</td>
<td>Uicontextmenu object's parent</td>
<td>Value: scalar figure handle</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
<td>Value: string</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
<td>Value: string (read-only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: uicontrol</td>
</tr>
</tbody>
</table>
These statements define a context menu associated with a line. When the user extend-clicks 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 extend-clicks on the line, the context menu appears, as shown in this figure:

See Also  
uibuttongroup, uicontrol, uimenu, uipanel
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.

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>
<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>
**UiContextMenu Properties**

**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

This property has no effect on `uicontextmenu` objects.

**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 uimenus defined for the uicontextmenu.

**Clipping**  
{on} | off

This property has no effect on `uicontextmenu` objects.

**CreateFcn**  
string

*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(gcbo,...
    'Visible','on'))
```

creates a default CreateFcn callback that runs whenever you create a new context menu. It sets the default Visible property of a context menu.

To override this default and create a context menu whose Visible property is set to a different value, call uicontextmenu with code similar to

```matlab
hpt = uicontextmenu(...,'CreateFcn','set(gcbo,...
    'Visible','off'))
```

**Note** To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uicontextmenu call. In the example above, if instead of redefining the CreateFcn property for this uicontextmenu, you had explicitly set Visible to off, the default CreateFcn callback would have set Visible back to the default, i.e., on.

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

**DeleteFcn** string

*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.

**HitTest** {on} | off

This property has no effect on `uicontextmenu` objects.

**Interruptible** {on} | off

*Callback routine interruption mode.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:
Uicontextmenu Properties

- 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

*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** vector

*Uicontextmenu's position.* A two-element vector that defines the location of a context menu posted by setting the Visible property value to on. Specify Position as

```
[x y]
```
where vector elements represent the horizontal and vertical distances in pixels from the bottom left corner of the figure window, panel, or button group to the top left corner of the context menu.

**Selected**  
please | off
This property has no effect on uicontextmenu objects.

**SelectionHighlight**  
please | off
This property has no effect on uicontextmenu objects.

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

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

**UIContextMenu**  
handle
This property has no effect on uicontextmenus.

**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**  
please | 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
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 “Properties” on page 2-2332 for information about these and other properties.

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
- 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

- Setting Component Properties — the Property Inspector
- Programming Callbacks for GUI Components

**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.

- `'listbox'` – List boxes display a list of items (defined using the `String` property) 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 open to display a list of choices (defined using the `String` property) 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. You must specify a
value for the String property.

- '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 the mutually exclusive behavior
of 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.

- 'toggle' – 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.
### Properties

This table lists all properties useful for `uicontrol` objects, grouping them by function. Each property name acts as a link to a description of the property.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controlling Style and Appearance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BackgroundColor</td>
<td>Object background color</td>
<td>Value: <code>ColorSpec</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: system dependent</td>
</tr>
<tr>
<td>CData</td>
<td>Truecolor image displayed on the control</td>
<td>Value: matrix</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Color of text</td>
<td>Value: <code>ColorSpec</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: <code>[0 0 0]</code></td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Object highlighted when selected</td>
<td>Value: <code>on</code>, <code>off</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: <code>on</code></td>
</tr>
<tr>
<td>String</td>
<td>Uicontrol label, also list box and</td>
<td>Value: string</td>
</tr>
<tr>
<td></td>
<td>pop-up menu items</td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td>Uicontrol visibility</td>
<td>Value: <code>on</code>, <code>off</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: <code>on</code></td>
</tr>
<tr>
<td><strong>General Information About the Object</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children</td>
<td>Uicontrol objects have no children</td>
<td></td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uicontrol</td>
<td>Value: <code>on</code>, <code>inactive</code>, <code>off</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: <code>on</code></td>
</tr>
<tr>
<td>Parent</td>
<td>Uicontrol object’s parent</td>
<td>Value: figure, uipanel, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uibuttongroup handle</td>
</tr>
<tr>
<td>Selected</td>
<td>Whether object is selected</td>
<td>Value: <code>on</code>, <code>off</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: <code>off</code></td>
</tr>
<tr>
<td>SliderStep</td>
<td>Slider step size</td>
<td>Value: two-element vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: <code>[0.01 0.1]</code></td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Property Value</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>Style</strong></td>
<td>Type of uicontrol object</td>
<td>Value: pushbutton, togglebutton, radiobutton, checkbox, edit, text, slider, listbox, popupmenu Default: pushbutton</td>
</tr>
<tr>
<td><strong>Tag</strong></td>
<td>User-specified object identifier</td>
<td>Value: string</td>
</tr>
<tr>
<td><strong>TooltipString</strong></td>
<td>Content of object's tooltip</td>
<td>Value: string</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Class of graphics object</td>
<td>Value: string (read-only) Default: uicontrol</td>
</tr>
<tr>
<td><strong>UserData</strong></td>
<td>User-specified data</td>
<td>Value: matrix</td>
</tr>
</tbody>
</table>

**Controlling the Object Position**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Position</strong></td>
<td>Size and location of uicontrol object</td>
<td>Value: position rectangle Default: [20 20 60 20]</td>
</tr>
<tr>
<td><strong>Units</strong></td>
<td>Units to interpret position vector</td>
<td>Value: pixels, normalized, inches, centimeters, points, characters Default: pixels</td>
</tr>
</tbody>
</table>

**Controlling Fonts and Labels**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FontAngle</strong></td>
<td>Character slant</td>
<td>Value: normal, italic, oblique Default: normal</td>
</tr>
<tr>
<td><strong>FontName</strong></td>
<td>Font family</td>
<td>Value: string Default: system dependent</td>
</tr>
<tr>
<td><strong>FontSize</strong></td>
<td>Font size</td>
<td>Value: size in FontUnits Default: system dependent</td>
</tr>
</tbody>
</table>
### uiControl

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FontUnits</td>
<td>Font size units</td>
<td>Value: points, normalized, inches, centimeters, pixels</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: points</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Weight of text characters</td>
<td>Value: light, normal, demi, bold</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: normal</td>
</tr>
<tr>
<td>HorizontalAlignment</td>
<td>Alignment of label string</td>
<td>Value: left, center, right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: depends on uicontrol object</td>
</tr>
<tr>
<td>String</td>
<td>Uicontrol object label, also list box and pop-up menu items</td>
<td>Value: string</td>
</tr>
</tbody>
</table>

**Controlling Callback Routine Execution**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyAction</td>
<td>Callback routine interruption</td>
<td>Value: cancel, queue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: queue</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Button-press callback routine</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>Callback</td>
<td>Control action</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>KeyPressFcn</td>
<td>Key press callback routine</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>UIContextMenu</td>
<td>Uicontextmenu object associated with the uicontrol</td>
<td>Value: handle</td>
</tr>
</tbody>
</table>
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
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:

```matlab
set(0,'DefaultUicontrolPropertyName',PropertyValue...)
set(gcf,'DefaultUicontrolPropertyName',PropertyValue...)
```

where `PropertyName` 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

- Setting Component Properties — the Property Inspector
- Programming Callbacks for GUI Components

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>CData</td>
<td>Truecolor image displayed on the control</td>
</tr>
<tr>
<td>Children</td>
<td>Uicontrol objects have no children</td>
</tr>
</tbody>
</table>
# Uicontrol 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 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>
<tr>
<td>Selected</td>
<td>Whether object is selected</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Object highlighted when selected</td>
</tr>
</tbody>
</table>
Uicontrol Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SliderStep</td>
<td>Slider step size</td>
</tr>
<tr>
<td>String</td>
<td>Uicontrol label, also list box and pop-up menu items</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 executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uicontrol. When the uicontrol’s `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the mouse 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 on a push button).

`Callback` string (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.
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, or
- For a multiline editable text box, press Ctrl+Enter.

Callback routines defined for static text do not execute because no action is associated with these objects.

**CData** matrix

*Truecolor image displayed on control.* A three-dimensional matrix 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.

**Children** matrix

The empty matrix; uicontrol objects have no children.

**Clipping** {on} | off

This property has no effect on uicontrols.

**CreateFcn** string

*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


**Uicontrol Properties**

```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

*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 `uitoggletool` whose `Enable` property is off, or when you right-click a uicontrol whose `Enable` property has any value, MATLAB performs these actions in this order:

1. Sets the figure’s `SelectionType` property.
2. Sets the figure’s `CurrentPoint` property.
3. Executes the figure’s `WindowButtonDownFcn` callback.

**Extent** position rectangle (read only)

Size of uicontrol character string. A four-element vector that defines the size and position of the character string used to label the uicontrol. It has the form:

\[ [0,0,width,height] \]

The first two elements are always zero. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

Since the `Extent` property is defined in the same units as the uicontrol itself, you can use this property to determine proper sizing for the uicontrol with regard to its label. Do this by

- Defining the `String` property and selecting the font using the relevant properties.
- Getting the value of the `Extent` property.
- Defining the `width` and `height` of the `Position` property to be somewhat larger than the `width` and `height` of the `Extent`.

For multiline strings, the `Extent` rectangle encompasses all the lines of text. For single line strings, the `Extent` is returned as a single line, even if the string wraps when displayed on the control.
**Uicontrol Properties**

**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):

```matlab
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

*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).
**UiControl Properties**

**FontWeight**  
*light | {normal} | demi | bold*

*Weight of text characters.* MATLAB uses this property to select a font from those available on your particular system. Setting this property to **bold** causes MATLAB to use a bold version of the font, when it is available on your system.

**ForegroundColor**  
*ColorSpec*

*Color of text.* This property determines the color of the text defined for the **String** property (the **uicontrol** label). Specify a color using a three-element RGB vector or one of 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 **CurrentUser** 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.
Uicontrol Properties

**Note** Radio buttons and toggle buttons that are managed by a uibuttongroup should not be accessed outside the button group. Set the HandleVisibility of such radio buttons and toggle buttons to off to prevent inadvertent access.

**HitTest** {on} | off

*Selectable by mouse click.* This property has no effect on uicontrol objects.

**HorizontalAlignment** left | {center} | right

*Horizontal alignment of label string.* This property determines the justification of the text defined for the String property (the uicontrol label):

- **left** — Text is left justified with respect to the uicontrol.
- **center** — Text is centered with respect to the uicontrol.
- **right** — Text is right justified with respect to the uicontrol.

On Microsoft Windows systems, this property affects only edit and text uicontrols.

**Interruptible** {on} | off

*Callback routine interruption mode.* If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute

If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. Whenever the callback calls one of the drawnow, figure, getframe, pause, or waitfor functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note
The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure's WindowButtonDownFcn callback routine, or an object's ButtonDownFcn or Callback routine are processed according to the rules described above.

KeyPressFcn  string or function handle

Key press callback function. A callback routine invoked by a key press when the callback’s 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>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Character interpretation of the key that was pressed.</td>
<td>'a'</td>
<td>'='</td>
</tr>
<tr>
<td>Modifier</td>
<td>Current modifier, such as 'control', or an empty cell array if there is no modifier</td>
<td>{1x0 cell}</td>
<td>{1x0 cell}</td>
</tr>
<tr>
<td>Key</td>
<td>Name of the key that was pressed.</td>
<td>'a'</td>
<td>'equal'</td>
</tr>
</tbody>
</table>

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

**ListboxTop**  
Scalar

*Index of top-most string displayed in list box.* This property applies only to the listbox style of uicontrol. It specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. *ListboxTop* is an index into the array of strings defined by the *String* property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

**Max**  
Scalar

*Maximum value.* This property specifies the largest value allowed for the *Value* property. Different styles of uicontrols interpret *Max* differently:

- Check boxes – *Max* is the setting of the *Value* property while the check box is selected.
• Editable text – 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.
• 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.
Uicontrol Properties

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.

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
twicontrol('Style','slider','Min',1,'Max',7,'
'SliderStep',[0.1 0.6])
```

clicking on the arrow button moves the indicator by,

```matlab
0.1*(7-1)
an =
0.6000
```

and clicking in the trough moves the indicator by,

```matlab
0.6*(7-1)
an =
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**

*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.

For **uicontrol objects that display only one line of text**, 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, each cell of a cell array of strings, and after any `\n` characters embedded in the string. Vertical slash (`'\'|`) 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 items as a cell array of strings, a padded string matrix, or within a string vector separated
by vertical slash (`|`) characters. Use the Value property to set the index of the initial item selected.

For editable text, this property value is set to the string entered by the user.

### Setting the String Property to a Reserved Word

Setting a property value to default, remove, or factory produces the effect described in Setting Default Values. To set a property to one of these words (e.g., String property set to the word `Default`), you must precede the word with the backslash character. For example,

```matlab
h = uicontrol('Style','edit','String','\Default');
```

<table>
<thead>
<tr>
<th>Style</th>
<th>{pushbutton}</th>
<th>togglebutton</th>
<th>radiobutton</th>
<th>checkbox</th>
<th>edit</th>
<th>text</th>
<th>slider</th>
<th>listbox</th>
<th>popupmenu</th>
</tr>
</thead>
</table>

*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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>string (GUIDE sets this property)</th>
</tr>
</thead>
</table>

*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.

<table>
<thead>
<tr>
<th>TooltipString</th>
<th>string</th>
</tr>
</thead>
</table>

*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.

<table>
<thead>
<tr>
<th>Type</th>
<th>string (read only)</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>UIContextMenu</th>
<th>handle</th>
</tr>
</thead>
</table>

*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.
UiControl Properties

**Units**

{pixels} | normalized | inches | centimeters | points | characters
(GUIDE default: normalized)

*Units of measurement.* MATLAB uses these units to interpret the Extent and Position properties. All units are measured from the lower-left corner of the parent object.

- **Normalized** units map the lower-left corner of the parent object to (0,0) and the upper-right corner to (1.0,1.0).
- **Pixels**, **inches**, **centimeters**, and **points** are absolute units (1 point = 1/72 inch).
- **Character** units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

**UserData**

*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.
**Uicontrol Properties**

- 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.
**Purpose**
Standard dialog box for selecting a directory

**Syntax**

directory_name = uigetdir

directory_name = uigetdir('start_path')

directory_name = uigetdir('start_path','dialog_title')

directory_name = uigetdir('start_path','dialog_title',x,y)

**Description**

uigetdir displays a dialog box enabling the user to browse through the directory structure and select a directory.

directory_name = uigetdir opens a dialog box in the current directory displaying the default title.

directory_name = uigetdir('start_path') opens a dialog box in the directory specified by start_path.

directory_name = uigetdir('start_path','dialog_title') opens a dialog box with the specified title.

directory_name = uigetdir('start_path','dialog_title',x,y) positions the dialog box at position [x,y], where x and y are the distance in pixel units from the left and top edges of the screen. This feature is only supported on UNIX platforms.

**Remarks**

**Returned directory_name**

directory_name is a string containing the path to the directory selected in the dialog box. If the user presses the Cancel button or closes the dialog window, directory_name is returned as the number 0.

**Specifying start_path**

start_path specifies the directory to display when the dialog is first opened. If start_path is a string representing 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 working directory.
If `start_path` is not a valid directory path, the dialog box opens in the base directory:

- On Windows systems, the base directory is the Windows desktop directory.
- On UNIX systems, the base directory is the directory from which MATLAB is started.

**Specifying dialog_title**
The placement of `dialog_title` in the dialog box depends on the computer system:

- On Windows systems, the string replaces the default caption inside the dialog box for specifying instructions to the user.
- On UNIX systems, the string replaces the default title of the dialog box.

If you do not specify the `dialog_title` argument, MATLAB uses the default string `Select Directory to Open`.

**Adding and Moving Directories**
On Windows systems, 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.

**Examples**
**Example 1**
The following statement displays the directories on the C: drive.

```matlab
dname = uigetdir('C:\');```
The dialog box is shown in the following figure.

If the user selects the directory `MATLAB6.5\help\techdoc`, as shown in the figure, and clicks **OK**, `uigetdir` returns

```
dname = C:\MATLAB6.5\help\techdoc
```

**Example 2**

The following statement uses the `matlabroot` command to display the MATLAB root directory in the dialog box:

```
uigetdir(matlabroot,'MATLAB Root Directory')
```
If the user selects the directory MATLAB6.5\notebook\pc, as shown in the figure, uigetdir returns a string like

C:\MATLAB6.5\notebook\pc

assuming that MATLAB is installed on drive C:\.

See Also  uigetfile, uiputfile
Purpose
Standard dialog box for retrieving files

Syntax
ugifile
ugifile('FilterSpec')
ugifile('FilterSpec','DialogTitle')
ugifile('FilterSpec','DialogTitle','DefaultName')
ugifile(...,'Location',[x y])
ugifile(...,'MultiSelect',selectmode)
[FileName,PathName] = uigifile(...)
[FileName,PathName,FilterIndex] = uigifile(...)

Description
ugifile displays a dialog box used to retrieve one or more files. The dialog box lists the files and directories in the current directory.

ugifile('FilterSpec') displays a dialog box that lists files in the current directory. FilterSpec determines the initial display of files and can include the * wildcard. For example, '* .m' lists all the MATLAB M-files.

If FilterSpec is a string or cell array, uigifile appends 'All Files' to the list of file types. If FilterSpec is a cell array, the first column contains the list of extensions, and the second column contains the list of descriptions. FilterSpec 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 not specified, uigifile uses the default list of file types (i.e., all MATLAB files).

ugifile('FilterSpec','DialogTitle') displays a dialog box that has the title DialogTitle.

ugifile('FilterSpec','DialogTitle','DefaultName') displays a dialog box in which a specified string, in this case 'DefaultName', appears in the File name field. 'Default Name' can be a filename or the name of a directory. If it is the name of a directory, you must follow it with a slash (/) or backslash (\) separator.

ugifile(...,'Location',[x y]) positions the dialog box at position [x,y], where x and y are the distances in pixel units from the left and top edges of the screen. This feature is supported only on UNIX platforms.
uigetfile(...,'MultiSelect',selectmode) specifies if multiple file selection is enabled for the uigetfile dialog. Valid values for selectmode are 'on' and 'off' (default). If the value of '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. Otherwise, Filename is a string representing the selected filename. Because multiple selections are always in the same directory, PathName is always a string that represents a single directory.

[FileName,PathName] = uigetfile(...) returns the name and path of the file selected in the dialog box. After the user clicks the Done button, FileName contains the name of the file selected and PathName contains the name of the path selected. If the user clicks the Cancel button or closes the dialog window, FileName and PathName are set to 0.

[FileName,PathName,FilterIndex] = uigetfile(...) returns the index of the filter selected in the dialog box. The indexing starts at 1. If the user clicks the Cancel button or closes the dialog window, FilterIndex is set to 0.

**Remarks**

A successful return occurs only if all the selected files exist. If the user selects a file that does not exist, an error message is displayed and control returns to the dialog box.

**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('*', '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({"*.m;*.mdl;*.mat;*.*"}, 'File Selector');
```
Example 3. If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the 'MATLAB Files' description.

```matlab
[filename, pathname] = uigetfile(...
{'*.m;*.fig;*.mat;*.mdl','MATLAB Files (*.m,*.fig,*.mat,*.mdl)';
 ' *.m', 'M-files (*.m)'; ... 
 ' *.fig', 'Figures (*.fig)'; ... 
 ' *.mat', 'MAT-files (*.mat)'; ... 
 ' *.mdl', 'Models (*.mdl)'; ... 
 ' *.*', 'All Files (*.*)'}, ... 
 'Pick a file');
```

The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the 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.

![Image of uigetfile dialog box](image)

Example 4. The following code checks for the existence of the file and displays a message about the result of the open operation.
[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');
uigetfile

See Also  uiputfile, uigetdir
**Purpose**
Open Import Wizard, the graphical user interface 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
Purpose
Create menus on figure windows

Syntax
uimenu('PropertyName',PropertyValue,...)
uimenu(parent,'PropertyName',PropertyValue,...)
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 can 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.

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. 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 menu item. uimenu optionally returns the handle to the created uimenu object.

Uimenu children exist in figures whose WindowStyle is normal. If a figure containing uimenu children is changed to WindowStyle 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 location of the uimenu on the figure menu bar. When MenuBar is figure, a set of built-in menus precedes the uimenus on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user). When MenuBar is none, uimenus 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`.

**Properties**

This table lists all properties useful to `uimenu` objects, grouping them by function. Each property name acts as a link to a description of the property.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controlling Style and Appearance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checked</td>
<td>Menu check indicator</td>
<td>Value: on, off; Default: off</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Color of text</td>
<td>Value: ColorSpec; Default: [0 0 0]</td>
</tr>
<tr>
<td>Label</td>
<td>Menu label</td>
<td>Value: string</td>
</tr>
<tr>
<td>Separator</td>
<td>Separator line mode</td>
<td>Value: on, off; Default: off</td>
</tr>
<tr>
<td>Visible</td>
<td>Uimenu visibility</td>
<td>Value: on, off; Default: on</td>
</tr>
<tr>
<td><strong>General Information About the Object</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accelerator</td>
<td>Keyboard equivalent</td>
<td>Value: character</td>
</tr>
<tr>
<td>Children</td>
<td>Handles of submenus</td>
<td>Value: vector of handles</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uimenu</td>
<td>Value: on, off; Default: on</td>
</tr>
<tr>
<td>Parent</td>
<td>Uimenu object's parent</td>
<td>Value: handle</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
<td>Value: string</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
<td>Value: string (read-only); Default: uimenu</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
<td>Value: matrix</td>
</tr>
</tbody>
</table>

**Controlling the Object Position**
**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
You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the inspect function at the command line.
- The set and get commands enable you to set and query the values of properties

You can set default Uimenu properties on the root, figure and menu levels:

```
set(0,'DefaultUimenuPropertyName',PropertyValue...)
set(gcf,'DefaultUimenuPropertyName',PropertyValue...)
set(menu_handle,'DefaultUimenuPropertyName',PropertyValue...)
```

Where `PropertyName` is the name of the Uimenu property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of property see Setting Default Property Values.

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>
<tr>
<td>CreateFcnc</td>
<td>Callback routine executed during object creation</td>
</tr>
<tr>
<td>DeleteFcnc</td>
<td>Callback routine executed during object deletion</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uimenu</td>
</tr>
</tbody>
</table>
Accelerator  character

**Keyboard equivalent.** A 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

*Menu action.* A callback routine that executes whenever you select the menu. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

A menu with children (submenus) executes its callback routine before displaying the submenus. A menu without children executes its callback routine when you release the mouse button (i.e., on the button up event).

**Checked** on | {off}

*Menu check indicator.* Setting this property to on places a check mark next to the corresponding menu item. Setting it to off removes the check mark. You can use this feature to create menus that indicate the state of a particular option. For example, suppose you have a menu item called **Show axes** that toggles the visibility of an axes between visible and invisible each time the user selects the menu item. If you want a check to appear next to the menu item
when the axes are visible, add the following code to the callback for the Show axes menu item:

```matlab
if strcmp(get(gcbo, 'Checked'), 'on')
    set(gcbo, 'Checked', 'off');
else
    set(gcbo, 'Checked', 'on');
end
```

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. Also, this property does not display the check mark on top level menus or submenus, although you can change the value of the property for these menus.

Note the following platform differences:

- On UNIX, the check mark is not displayed on submenus that have submenus.
- On Windows, the check mark is displayed on submenus, whether or not they have submenus.

**Children**

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
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.
Uimenu Properties

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  {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 using the “&” character. Whatever character follows the “&” in the string appears underlined and selects the menu item when you type 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:

- `Open selection` yields **Open selection**
- `Save & Go` yields **Save & Go**
Uimenu Properties

Parent

The handle of the uimenu’s parent object. The parent of a uimenu object is the figure on whose menu bar it displays, or the uimenu of which it is a submenu. You can move a uimenu object to another figure by setting this property to the handle of the new parent.

Position scalar

Relative menu position. The value of Position indicates placement on the menu bar or within a menu. Top-level menus are placed from left to right on the menu bar according to the value of their Position property, with 1 representing the left-most position. The individual items within a given menu are placed from top to bottom according to the value of their Position property, with 1 representing the top-most position.

Separator on | {off}

Separator line mode. Setting this property to on draws a dividing line above the menu item.

Tag string

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

Type string (read only)

Class of graphics object. For uimenu objects, Type is always the string ‘uimenu’.

UserData matrix

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

Visible {on} | off

Uimenu visibility. By default, all uimenus are visible. When set to off, the uimenu is not visible, but still exists and you can query and set its properties.
**Purpose**
Convert to unsigned integer

**Syntax**

- `I = uint8(X)`
- `I = uint16(X)`
- `I = uint32(X)`
- `I = uint64(X)`

**Description**

`I = uint*(X)` converts the elements of array `X` into unsigned integers. `X` can be any numeric object (such as a `double`). The results of a `uint*` operation are shown in the next table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output Range</th>
<th>Output Type</th>
<th>Bytes per Element</th>
<th>Output Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8</code></td>
<td>0 to 255</td>
<td>Unsigned 8-bit integer</td>
<td>1</td>
<td><code>uint8</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td>0 to 65,535</td>
<td>Unsigned 16-bit integer</td>
<td>2</td>
<td><code>uint16</code></td>
</tr>
<tr>
<td><code>uint32</code></td>
<td>0 to 4,294,967,295</td>
<td>Unsigned 32-bit integer</td>
<td>4</td>
<td><code>uint32</code></td>
</tr>
<tr>
<td><code>uint64</code></td>
<td>0 to 18,446,744,073,709,551,615</td>
<td>Unsigned 64-bit integer</td>
<td>8</td>
<td><code>uint64</code></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`:

```
intmin('uint32')          intmax('uint32')
ans =                     ans =
0                          4294967295
```

*See Also*

double, single, int8, int16, int32, int64, intmax, intmin
Purpose
Display a file selection dialog with appropriate file filters

Syntax
uiopen

Description
uiopen displays a 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** 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.

**See Also** uigetfile, uiputfile, uisave
 UIPanel

Purpose

Uipanel container object

Syntax

h = uipanel('PropertyName1',value1,'PropertyName2',value2,...)

Description

A uipanel groups components. It can contain user interface controls with which
the user interacts directly. It can also contain axes, other uipanels, and
uibuttongroups. It cannot contain ActiveX controls.

h = uipanel 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.

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.

Properties

This table lists all properties useful to uipanel objects, grouping them by
function. Each property name acts as a link to a description of the property.
Curly braces denote the default value, if any

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controlling Style and Appearance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BackgroundColor</td>
<td>Color of the uipanel background</td>
<td>ColorSpec. Default is the same as the default uicontrol background.</td>
</tr>
<tr>
<td>BorderType</td>
<td>Type of border around the uipanel area.</td>
<td>[none</td>
</tr>
<tr>
<td>BorderWidth</td>
<td>Width of the panel border.</td>
<td>Integer. Default is 1.</td>
</tr>
</tbody>
</table>
**Property Name** | **Description** | **Property Value**
--- | --- | ---
Clipping | Clipping of child axes, uipanels, and uibuttongroups to the uipanel. Does not affect child uicontrols. | [{on}|off]
ForegroundColor | Title font color and/or color of 2-D border line | ColorSpec. Default is [0 0 0] (black).
HighlightColor | 3-D frame highlight color | ColorSpec. Default is [1 1 1] (white).
SelectionHighlight | Object highlighted when selected | [{on}|off]
ShadowColor | 3-D frame shadow color | ColorSpec. Default is [.5 .5 .5] (grey).
Visible | Uipanel visibility. Note: Controls the Visible property of child axes, uibuttongroups, and uipanels. Does not affect child uicontrols. | [{on}|off]

**General Information About the Object**

| **Children** | All children of the uipanel | Vector of handles |
| **Parent** | Uipanel object's parent | Figure, uipanel, or uibuttongroup handle |
| **Selected** | Whether object is selected | [{on}|{off}]
| **Tag** | User-specified object identifier | String |
| **UserData** | User-specified data | Matrix |
### Controlling the Object Position

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>Panel position relative to parent figure, uipanel, or uibuttongroup</td>
<td>Position spec ([x \ y \ w \ h]). Default is ([0 \ 0 \ 1 \ 1])</td>
</tr>
<tr>
<td>Units</td>
<td>Units used to interpret the position vector</td>
<td>([\text{inches}</td>
</tr>
</tbody>
</table>

### Controlling Fonts and Labels

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FontAngle</td>
<td>Title font angle</td>
<td>([{\text{normal}}</td>
</tr>
<tr>
<td>FontName</td>
<td>Title font name</td>
<td>String. Default is system dependent.</td>
</tr>
<tr>
<td>FontSize</td>
<td>Title font size</td>
<td>Integer. Default is system dependent.</td>
</tr>
<tr>
<td>FontUnits</td>
<td>Title font units</td>
<td>([\text{inches}</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Title font weight</td>
<td>([\text{light}</td>
</tr>
<tr>
<td>Title</td>
<td>Title string</td>
<td>String</td>
</tr>
<tr>
<td>TitlePosition</td>
<td>Location of title string in relation to the panel</td>
<td>([{\text{lefttop}}</td>
</tr>
</tbody>
</table>

### Controlling Callback Routine Execution

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusyAction</td>
<td>Interruption of other callback routines</td>
<td>([{\text{queue}}</td>
</tr>
<tr>
<td>ButtonDownFcn</td>
<td>Button-press callback routine</td>
<td>String or function handle</td>
</tr>
</tbody>
</table>
**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]);
hsbp = uicontrol('Parent',hsp,'String','Push here',...
    'Position', [18 18 72 36]);
```
See Also  hgtransform, uibuttongroup, uicontrol
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.

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>
<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>
</tbody>
</table>
## Uipanel Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>HitTest</td>
<td>Selectable by mouse click</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
</tr>
<tr>
<td>Parent</td>
<td>Uipanel object’s parent</td>
</tr>
<tr>
<td>Position</td>
<td>Panel position relative to parent figure or uipanel</td>
</tr>
<tr>
<td>ResizeFcn</td>
<td>User-specified resize routine</td>
</tr>
<tr>
<td>Selected</td>
<td>Whether object is selected</td>
</tr>
<tr>
<td>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>
</tbody>
</table>
### Uipanel Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIContextMenu</td>
<td>Associates uicontext menu 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. Note: Controls the Visible property of child axes, uibuttongroups and uipanels. Does not affect child uicontrols.</td>
</tr>
</tbody>
</table>

**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.

HitTest  {on} | off

Selectable by mouse click. HitTest determines if the figure 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 figure. If HitTest is off, clicking the figure sets the CurrentObject to the empty matrix.

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

```
[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.

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 figure 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'.

```matlab
h = findobj(figurehandles,'Tag','FormatTb')
```

**Title**  
*string*

*Title string.* The text displayed in the panel title. You can position the title using the *TitlePosition* property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('|') characters are not interpreted as line breaks and instead show up in the text displayed in the uipanel title.

Setting a property value to *default*, *remove*, or *factory* produces the effect described in Setting Default Values. To set *Title* to one of these words, you must precede the word with the backslash character. For example,

```matlab
hp = uipanel(...,'Title','\Default');
```
**Uipanel Properties**

**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.

**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. All units are measured from the lower-left corner of the figure window.

- **Normalized** units map the lower-left corner of the 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 a push button on the current or specified toolbar

**Syntax**

```matlab
htt = uipushtool('PropertyName1',value1,...
    'PropertyName2',value2,...)
htt = uipushtool(ht,...)
```

**Description**

`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(htt)` to see a list of `uipushtool` object properties and their current values. Type `set(htt)` 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.

`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 only appear in figures whose `WindowStyle` is `normal`. If a figure containing a uitoolbar and its uipushtool children is changed to `WindowStyle modal`, the uipushtools still exist and are contained in the `Children` list of the uitoolbar, but are not displayed until the `WindowStyle` is changed to `normal`.

**Properties**

This table lists all properties useful to `uipushtool` objects, grouping them by function. Each property name acts as a link to a more detailed description of the property.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controlling Style and Appearance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CData</td>
<td>Truecolor image displayed on the uipushtool</td>
<td>Value: m-by-n-by-3 array</td>
</tr>
</tbody>
</table>
### uipushtool

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separator</td>
<td>Separator line mode</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Visible</td>
<td>Uipushtool visibility</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>General Information About the Object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BeingDeleted</td>
<td>This object is being deleted</td>
<td>Value: on, off (read-only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uipushtool</td>
<td>Value: on, inactive, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>Parent</td>
<td>Uipushtool object’s parent toolbar.</td>
<td>Value: handle</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
<td>Value: string</td>
</tr>
<tr>
<td>TooltipString</td>
<td>Content of object’s tooltip</td>
<td>Value: string</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
<td>Value: string (read-only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: uipushtool</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
<td>Value: array</td>
</tr>
<tr>
<td>Controlling Callback Routine Execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BusyAction</td>
<td>Interruption of other callback routines</td>
<td>Value: cancel, queue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: queue</td>
</tr>
<tr>
<td>ClickedCallback</td>
<td>Control action.</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>CreateFcn</td>
<td>Callback routine executed during object creation</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Callback routine executed during object deletion</td>
<td>Value: string or function handle</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
</tbody>
</table>
 Examples

This example creates a uitoolbar object and places a uipushtool object on it.

```matlab
h = figure('ToolBar','none')
ht = uitoolbar(h)
a = [.05:.05:0.95];
b(:,:,1) = repmat(a,19,1)';
b(:,:,2) = repmat(a,19,1);
b(:,:,3) = repmat(flipdim(a,2),19,1);
hpt = uipushtool(ht,'CData',b,'TooltipString','Hello')
```

### Controlling Access to Objects

**HandleVisibility**

Handle accessibility from command line and code associated with the GUIs.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| HandleVisibility | Handle accessibility from command line and code associated with the GUIs. | Value: on, callback, off
|                | Default: on          |                |

![Figure No. 1](image.png)
uipushtool

Object Hierarchy

See Also get, set, uicontrol, uitoggletool, uicontrol
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:

```matlab
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>
<tr>
<td>Enable</td>
<td>Enable or disable the uipushtool.</td>
</tr>
<tr>
<td>HandleVisibility</td>
<td>Control access to object's handle.</td>
</tr>
<tr>
<td>Interruptible</td>
<td>Callback routine interruption mode.</td>
</tr>
<tr>
<td>Parent</td>
<td>Handle of uipushtool's parent.</td>
</tr>
</tbody>
</table>
Uipushtool Properties

<table>
<thead>
<tr>
<th>Property (Continued)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separator</td>
<td>Separator line mode</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object label.</td>
</tr>
<tr>
<td>TooltipString</td>
<td>Content of object’s tooltip.</td>
</tr>
<tr>
<td>Type</td>
<td>Object class.</td>
</tr>
<tr>
<td>UserData</td>
<td>User specified data.</td>
</tr>
<tr>
<td>Visible</td>
<td>Uipushtool visibility.</td>
</tr>
</tbody>
</table>

**BeingDeleted** on | {off} (read only)

*This object is being deleted.* The BeingDeleted property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB 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. The largest image that fits on the push tool is 20-by-20. If the array is larger, only the center 20-by-20 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

```matlab
ima(:,:,1) = rand(20);
ima(:,:,2) = rand(20);
ima(:,:,3) = rand(20);
set(0,'DefaultUipushtoolCreateFcn','set(gcbo,''Cdata'',ima)')
```

creates a default CreateFcn callback that runs whenever you create a new push tool. It sets the default image ima on the push tool.

To override this default and create a push tool whose Cdata property is set to a different image, call uipushtool with code similar to
Uipushtool Properties

\begin{verbatim}
a = [.05:.05:0.95];
ingb(:,:,1) = repmat(a,19,1)';
ingb(:,:,2) = repmat(a,19,1);
ingb(:,:,3) = repmat(flipdim(a,2),19,1);
hpt = uipushtool(...,'CreateFcn',set(gcbo,'CData',imgb)',...)
\end{verbatim}

**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 imaga.

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

**DeleteFcn** string or function handle

*Callback routine executed during object deletion.* A callback routine that executes when you delete the uipushtool object (e.g., when you call the delete function or cause the figure containing the uipushtool to reset). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

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

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

**Enable** {on} | off

*Enable or disable the uipushtool.* This property controls how uipushtools respond to mouse button clicks, including which callback routines execute.

- **on** – The uipushtool is operational (the default).
- **off** – The uipushtool is not operational and its image (set by the CData property) is grayed out.
When you left-click on a uipushtool whose Enable property is on, MATLAB performs these actions in this order:

1. Sets the figure’s SelectionType property.
2. Executes the push tool’s ClickedCallback routine.
3. Does not set the figure’s CurrentPoint property and does not execute the figure’s WindowButtonDownFcn callback.

When you left-click on a uipushtool whose Enable property is off, or when you right-click a uipushtool whose Enable property has any value, MATLAB performs these actions in this order:

1. Sets the figure’s SelectionType property.
2. Sets the figure’s CurrentPoint property.
3. Executes the figure’s WindowButtonDownFcn callback.
4. Does not execute the push tool’s ClickedCallback routine.

**HandleVisibility**  {on} | callback | off

*Control access to object’s handle.* This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure’s CurrentObject property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

- Handles are always visible when HandleVisibility is on.
- Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting HandleVisibility to off makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.
You can set the root ShowHiddenHandles property to on to make all handles visible, regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties.

**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.
**Uipushtool Properties**

**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(uitoolbarhandles,'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 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.
**uiputfile**

**Purpose**
Standard dialog box for saving files

**Syntax**

```
uiputfile
uiputfile('FilterSpec')
uiputfile('FilterSpec','DialogTitle')
uiputfile('FilterSpec','DialogTitle', 'DefaultName')
uiputfile(...,'Location',[x y])
[FileName,PathName] = uiputfile(...)  
[FileName,PathName,FilterIndex] = uiputfile(...)  
```

**Description**

`uiputfile` displays a dialog box used to select a file for saving. The dialog box lists the files and directories in the current directory.

`uiputfile('FilterSpec')` displays a dialog box that lists files in the current directory. `FilterSpec` determines what files are displayed initially in the dialog box. For example `'*'.m'` lists all MATLAB M-files.

If `FilterSpec` is a string or cell array, `uiputfile` appends `All Files` to the list of file types. If `FilterSpec` is a cell array, the first column is used as the list of extensions, and the second column is used as the list of descriptions. `FilterSpec` 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 not specified, `uiputfile` uses the default list of file types (i.e., all MATLAB files).

`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')
```

`uiputfile('FilterSpec','DialogTitle','DefaultName')` displays a dialog box in which a specified string, in this case `DefaultName`, appears in the File name field. `Default Name` can be a filename or the name of a directory. If it is the name of a directory, you must follow it with a slash (/) or backslash (\) separator.

`uiputfile(...,'Location',[x y])` positions the dialog box at screen position `[x,y]`, where `x` and `y` are the distances in pixel units from the left and top edges of the screen. This feature is supported only on UNIX platforms.
The `uiputfile` function returns the name and path of the file selected in the dialog box. If the user clicks the Cancel button or closes the dialog window, `FileName` and `PathName` are set to 0.

```matlab
[FileName, PathName] = uiputfile(...)
```

This function can also return the index of the filter selected in the dialog box. The indexing starts at 1. If the user clicks the Cancel button or closes the dialog window, `FilterIndex` is set to 0.

```matlab
[FileName, PathName, FilterIndex] = uiputfile(...)
```

**Remarks**

If the user specifies or selects an existing filename, a warning message is displayed asking whether the user wants to overwrite the file. If the user selects Yes, the existing file is overwritten and `uiputfile` returns successfully. If the user selects No, control returns to the dialog. For a successful return, the specified filename must be valid.

**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 (`*.*`).

```matlab
[file, path] = uiputfile('animinit.m', 'Save file name');
```

**Example 2.** The following statement displays a dialog box titled 'Save Workspace As' with the filter specifier set to MAT-files.

```matlab
[file, path] = uiputfile('*.mat', 'Save Workspace As');
```
Example 3. To display several file types in the **Save as type** list box, separate each file extension with a semicolon, as in the following code. Note that `uiputfile` displays a default description for each known file type, such as “Simulink Models” for .mdl files.

```matlab
[filename, pathname] = uiputfile(...
    {'*.m'; '*.mdl'; '*.mat'; '*.*'}, ...
    '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.

![Save as dialog box](image)

Example 5. The following code checks for the existence of the file and displays a message about the result of the open operation.
[filename, pathname] = uigetfile('*.*', 'Pick an M-file');
if isequal(filename,0) | isequal(pathname,0)
    disp('User selected Cancel')
else
    disp(['User selected', fullfile(pathname, filename)])
end

See Also

uigetfile
Purpose

Control program execution

Syntax

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 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.

See Also

`uicontrol`, `uimenu`, `waitfor`, `figure`, `dialog`
**uisave**

**Purpose**
Display a dialog for saving workspace variables

**Syntax**
uisave

**Description**
uisave displays a dialog for saving workspace variables to a MAT-file, as shown in the figure below.

If you type a name in the **File name** field, such as `my_vars`, and click **Save**, the dialog saves all workspace variables as `my_vars.mat`.

**Note**
uisave cannot be compiled. If you want to create a dialog that can be compiled, use uiputfile.

**See Also**
uigetfile, uiputfile, uiputfile
Purpose
Set an object's ColorSpec from a dialog box interactively

Syntax
\[ c = uisetcolor(h\_or\_c, 'DialogTitle'); \]

Description
uisetcolor displays a dialog box for the user to fill in, then applies the selected color to the appropriate property of the graphics object identified by the first argument.

h\_or\_c can be either a handle to a graphics object or an RGB triple. If you specify a handle, it must specify a graphics object that have a Color property. If you specify a color, it must be a valid RGB triple (e.g., \[1 0 0\] for red). The color specified is used to initialize the dialog box. If no initial RGB is specified, the dialog box initializes the color to black.

DialogTitle is a string that is used as the title of the dialog box.

c is the RGB value selected by the user. If the user presses Cancel from the dialog box, or if any error occurs, c is set to the input RGB triple, if provided; otherwise, it is set to 0.

See Also
ColorSpec
uisetfont

Purpose
Modify font characteristics for objects interactively

Syntax
uisetfont
uisetfont(h)
uisetfont(S)
uisetfont(h,'DialogTitle')
uisetfont(S,'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 the dialog box and returns the selected font properties.

uisetfont(h) displays a 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 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 dialog box with the title DialogTitle and returns the values of the font properties selected in the dialog box.

If a left-hand argument is specified, the properties FontName, FontUnits, FontSize, FontWeight, and FontAngle are returned 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:

h = text(.5,.5,'Figure Annotation');

2-2416
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:

% 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, save to d
d = uisetfont(c1);
% Apply those settings to c2
set(c2, d)

See Also
axestext, uicontrol
uistack

Purpose
Restack objects

Syntax
uistack(h)
uistack(h, stackopt)

Description
uistack enables you to change the stacking order of objects.

uistack(h, stackopt) moves 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, 'up', n) moves h up n steps
uistack(h, 'down', n) moves h down n steps

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)
```
uitoggletool

Purpose
Create a toggle button on the current or specified toolbar

Syntax
htt = uitoggletool('PropertyName1',value1,...
   'PropertyName2',value2,...)
htt = uitoggletool(ht,...)

Description
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.

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.

Uitoggletools only appear in figures whose WindowStyle is normal. If a figure
containing a uitoolbar and its uitoggletool children is changed to WindowStyle
modal, the uitoggletools still exist and are contained in the Children list of the
uitoolbar, but are not displayed until the WindowStyle is changed to normal.

Properties
This table lists all properties useful to uitoggletool objects, grouping them by
function. Each property name acts as a link to a more detailed description of
the property.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controlling Style and Appearance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CData</td>
<td>Truecolor image displayed on the uitoggletool</td>
<td>Value: m-by-n-by-3 array</td>
</tr>
<tr>
<td>Property Name</td>
<td>Property Description</td>
<td>Property Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>Separator</td>
<td>Separator line mode</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Visible</td>
<td>Uitoggletool visibility</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>BeingDeleted</td>
<td>This object is being deleted</td>
<td>Value: on, off (read-only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Enable</td>
<td>Enable or disable the uitoggletool</td>
<td>Value: on, inactive, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>Parent</td>
<td>Uitoggletool object’s parent toolbar</td>
<td>Value: handle</td>
</tr>
<tr>
<td>State</td>
<td>Uitoggletool state</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object identifier</td>
<td>Value: string</td>
</tr>
<tr>
<td>TooltipString</td>
<td>Content of object’s tooltip</td>
<td>Value: string</td>
</tr>
<tr>
<td>Type</td>
<td>Class of graphics object</td>
<td>Value: string (read-only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: uitoggletool</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
<td>Value: array</td>
</tr>
</tbody>
</table>

### General Information About the Object

- **BusyAction**: Interruption of other callback routines
  - Value: cancel, queue
  - Default: queue
- **ClickedCallback**: Control action independent of toggle tool position
  - Value: string or function handle
- **CreateFcn**: Callback routine executed during object creation
  - Value: string or function handle
- **DeleteFcn**: Callback routine executed during object deletion
  - Value: string or function handle
 Examples

This example creates a `uitoolbar` object and places a `uitoggletool` object on it.

```matlab
h = figure('ToolBar','none');
ht = uitoolbar(h);
a(:,:,1) = rand(20);
a(:,:,2) = rand(20);
a(:,:,3) = rand(20);
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello');
```
uitoggletool

Object Hierarchy

See Also

get, set, uicontrol, uipushtool, uitoolbar
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:

```matlab
set(h,'DefaultUitoggletoolPropertyName',PropertyValue...)
```

Where `h` can be the root handle (0), a figure handle, a uitoolbar handle, or a uitoggletool handle. `PropertyName` is the name of the Uitoggletool property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

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>
<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>
</tbody>
</table>
**Uitoggletool Properties**

<table>
<thead>
<tr>
<th>Property (Continued)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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 (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. The largest image that fits on the toggle tool is 20-by-20. If the array is larger, only the center 20-by-20 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''...',
```

2-2425
UiToggletool Properties

'abinet**)'

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.
Uitoggletool Properties

• 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:

1. Sets the figure’s SelectionType property.
2. Sets the figure’s CurrentPoint property.
3. Executes the figure’s WindowButtonDownFcn callback.
4. Does not execute the toggle tool’s OnCallback, OffCallback, or ClickedCallback routines.

HandleVisibility  {on} | callback | off

Control access to object’s handle. This property determines when an object’s handle is visible in its parent’s list of children. When a handle is not visible in its parent’s list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close. Neither is the handle visible in the parent figure’s CurrentObject property. Handles that are hidden are still valid. If you know an object’s handle, you can set and get its properties, and pass it to any function that operates on handles.

• Handles are always visible when HandleVisibility is on.
• Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
Uitoggletool Properties

- 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 interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's **Interruptible** property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement.
OffCallback string or function handle

*Control action.* A routine that executes if the uitoggletool's Enable property is set to on, and either

- The toggle tool State is set to off.
- The toggle tool is set to the off position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The ClickedCallback routine, if there is one, runs after the OffCallback routine runs to completion.

OnCallback string or function handle

*Control action.* A routine that executes if the uitoggletool's Enable property is set to on, and either

- The toggle tool State is set to on.
- The toggle tool is set to the on position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The ClickedCallback routine, if there is one, runs after the OffCallback routine runs to completion.

Parent handle

*Uitoggletool parent.* The handle of the uitoggletool's parent toolbar. You can move a uitoggletool object to another toolbar by setting this property to the handle of the new parent.

Separator on | {off}

*Separator line mode.* Setting this property to on draws a dividing line to left of the uitoggletool.

State on | {off}

*Uitoggletool state.* When the state is on, the toggle tool appears in the down, or pressed, position. When the state is off, it appears in the up position. Changing the state causes the appropriate OnCallback or OffCallback routine to run.

Tag string

*User-specified object identifier.* The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.
With the findobj function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the Tag value 'Bold'.

```matlab
h = findobj(uitoolbarhandles, '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 a toolbar on the current or specified 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 only appear in figures whose WindowStyle is normal or docked. If a figure containing a uitoolbar is changed to WindowStyle 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.

Properties
This table lists all properties useful to uitoolbar objects, grouping them by function. Each property name acts as a link to a more detailed description of the property.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visible</td>
<td>Uitoolbar visibility</td>
<td>Value: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
</tbody>
</table>
This example creates a figure with no toolbar, then adds a toolbar to it.

```matlab
h = figure('ToolBar','none')
ht = uitoolbar(h)
```
Object Hierarchy

See Also

set, get, uicontrol, uipushtool, uitoggletool
Uitoolbar 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 uitoolbar'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>
<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>
</tbody>
</table>
Uitoolbar Properties

<table>
<thead>
<tr>
<th>Property (Continued)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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.
Uitoolbar Properties

**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**

`handle`

UItoolbar parent. The handle of the uilobar’s parent figure. You can move a uilobar object to another figure by setting this property to the handle of the new parent.

**Tag**

`string`

User-specified object identifier. The Tag property provides a means to identify graphics objects with a user-specified label. You can define Tag as any string.
With the `findobj` function, you can locate an object with a given Tag property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the Tag value `'FormatTb'`.

```matlab
h = findobj(figurehandles,'Tag','FormatTb')
```

**Type**

String (read-only)

Object class. This property identifies the kind of graphics object. For uitoolbar objects, `Type` is always the string `'uitoolbar'`.

**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.
# undocheckout

## Purpose

Undo previous checkout from source control system

## Graphical Interface

As an alternative to the undocheckout function, use **Source Control Undo Checkout** in the Editor, Simulink, or Stateflow **File** menu.

## Syntax

```
undocheckout('filename')
undocheckout({'filename1','filename2','filename3', ...})
```

## 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. `filename` must be the full pathname for the file.

`undocheckout({'filename1','filename2','filename3', ...})` makes the `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 the files.

## Examples

Typing

```
undocheckout({'/matlab/mymfiles/clock.m', ...
    '/matlab/mymfiles/calendar.m'})
```

undoes the checkouts of `/matlab/mymfiles/clock.m` and `/matlab/mymfiles/calendar.m` from the source control system.

## See Also

`checkin`, `checkout`
**Purpose**
Set union of two vectors

**Syntax**

```matlab
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. The resulting vector is sorted in ascending order. In set theoretic terms, `C = A ∪ B`. `A` and `B` can be cell arrays of strings.

`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) ∪ b(ib)`, or for row combinations, `C = a(ia,:) ∪ 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.

**Examples**

```matlab
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 =
1     2     3     4```

**See Also**
intersect, setdiff, setxor, unique, ismember, issorted
**Purpose**

Unique elements of a vector

**Syntax**

```
b = unique(A)
b = unique(A, 'rows')
[b, m, n] = unique(...)
```

**Description**

`b = unique(A)` returns the same values as in `A` but with no repetitions. The resulting vector is sorted in ascending order. `A` can be a cell array of strings.

`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,:)`.
```

**Examples**

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

[b, m, n] = unique(A)
b =
    1 2 3 4 5 6 8 9
m =
    2 11 7 12 3 10 9 8
n =
    1 1 5 6 2 3 3 8 7 6 2 4

A(m)
ans =
    1 2 3 4 5 6 8 9

b(n)
ans =
    1 1 5 6 2 3 3 9 8 6 2 4
```
Because NaNs are not equal to each other, unique treats them as unique elements.

```plaintext
unique([1 1 NaN NaN])
ans =
    1  NaN  NaN
```

**See Also**

intersect, ismember, issorted, setdiff, setxor, union
**unix**

**Purpose**
Execute a 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') forces the output to the Command Window, even though it is also being assigned into a variable.

**Examples**
List all users that are currently logged in. It returns a zero (success) in s and a string containing the list of users in w.

```
[s,w] = unix('who');
```

The next example returns a nonzero value in s to indicate failure and returns an error message in w because why is not a UNIX command.

```
[s,w] = unix('why')
s = 1
w = why: Command not found.
```

When including the -echo flag, MATLAB displays the results of the command in the Command Window as it executes as well as assigning the results to the return variable, w.

```
[s,w] = unix('who','-echo');
```

**See Also**
dos, ! (exclamation point), perl, system
Purpose

Piecewise polynomial details

Syntax

\[ \text{[breaks,coefs,l,k,d]} = \text{unmkpp}(\text{pp}) \]

Description

\[ \text{[breaks,coefs,l,k,d]} = \text{unmkpp}(\text{pp}) \]
extracts, from the piecewise polynomial \( \text{pp} \), its breaks \( \text{breaks} \), coefficients \( \text{coefs} \), number of pieces \( \text{l} \), order \( \text{k} \), and dimension \( \text{d} \) of its target. Create \( \text{pp} \) using \text{spline} or the spline utility \text{mkpp}.

Examples

This example creates a description of the quadratic polynomial

\[ \text{as a piecewise polynomial } \text{pp}, \text{ then extracts the details of that description.} \]

\[
\text{pp} = \text{mkpp([-8 -4],[-1/4 1 0])};
\]

\[
\text{[breaks,coefs,l,k,d]} = \text{unmkpp}(\text{pp})
\]

\[
\text{breaks} = \\
-8 \quad -4
\]

\[
\text{coefs} = \\
-0.2500 \quad 1.0000 \quad 0
\]

\[
\text{l} = \\
1
\]

\[
\text{k} = \\
3
\]

\[
\text{d} = \\
1
\]

See Also

\text{mkpp, ppval, spline}
**unwrap**

**Purpose**
Correct phase angles to produce smoother phase plots

**Syntax**

\[
Q = \text{unwrap}(P) \\
Q = \text{unwrap}(P,\text{tol}) \\
Q = \text{unwrap}(P,[],\text{dim}) \\
Q = \text{unwrap}(P,\text{tol},\text{dim})
\]

**Description**

\(Q = \text{unwrap}(P)\) corrects the radian phase angles in a vector \(P\) by adding multiples of \(\pi\) when absolute jumps between consecutive elements of \(P\) are greater than the default jump tolerance of \(\pi\) radians. If \(P\) is a matrix, \text{unwrap} operates columnwise. If \(P\) is a multidimensional array, \text{unwrap} operates on the first nonsingleton dimension.

\(Q = \text{unwrap}(P,\text{tol})\) uses a jump tolerance \(\text{tol}\) instead of the default value, \(\pi\).

\(Q = \text{unwrap}(P,[],\text{dim})\) unwraps along \(\text{dim}\) using the default tolerance.

\(Q = \text{unwrap}(P,\text{tol},\text{dim})\) uses a jump tolerance of \(\text{tol}\).

**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 would result in a jump larger than the existing one, so \text{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.

\[
w = [0:.2:3,3.5:1:10]; \\
p = [0 \\
-1.5728 \\
-1.5747 \\
-1.5772 \\
-1.5790 \\
-1.5816 \\
-1.5852]
\]
Using `unwrap` to correct the phase angle, the resulting jump is 2.6959, which is less than the default jump tolerance. This figure plots the new curve over the original curve.

\[
\text{semilogx}(w, \text{unwrap}(p), 'r*-')
\]

**Note** If you have the Control System Toolbox, you can create the data for this example with the following code.

\[
\begin{align*}
\text{h} &= \text{freqresp}(\text{tf}(1,[1 .1 10 0])); \\
\text{p} &= \text{angle}(\text{h}(:));
\end{align*}
\]

**Example 2.** Array \( P \) features smoothly increasing phase angles except for discontinuities at elements (3,1) and (1,2).

\[
P = \begin{bmatrix}
0 & 7.0686 & 1.5708 & 2.3562 \\
0.1963 & 0.9817 & 1.7671 & 2.5525 \\
6.6759 & 1.1781 & 1.9635 & 2.7489
\end{bmatrix}
\]
The function \( Q = \text{unwrap}(P) \) eliminates these discontinuities.

\[
Q = 
\begin{array}{cccc}
0 & 7.0686 & 1.5708 & 2.3562 \\
0.1963 & 7.2649 & 1.7671 & 2.5525 \\
0.3927 & 7.4613 & 1.9635 & 2.7489 \\
0.5890 & 7.6576 & 2.1598 & 2.9452 \\
\end{array}
\]

**See Also**

abs, angle
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Extract contents of zip file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>unzip('zipfilename')</code></td>
</tr>
<tr>
<td></td>
<td><code>unzip('zipfilename','directory')</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>unzip('zipfilename')</code> extracts the contents of (unzips) the zip file named <code>zipfilename</code> into the current directory, where <code>zipfilename</code> was created using the <code>zip</code> function or any standard zip application. The path for <code>zipfilename</code> is relative to the current directory.</td>
</tr>
<tr>
<td></td>
<td><code>unzip('zipfilename','directory')</code> extracts the contents of the zip file named <code>zipfilename</code> into the specified directory. The paths for <code>zipfilename</code> and <code>directory</code> are relative to the current directory.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>Extract the contents of <code>d:/mymfiles/viewlet.zip</code>, putting the resulting files in the current directory.</td>
</tr>
<tr>
<td></td>
<td><code>unzip('d:/mymfiles/viewlet.zip')</code></td>
</tr>
<tr>
<td></td>
<td>Unzip the zip file <code>mymfiles</code> in the current directory, putting the resulting files in the directory <code>archives</code>, which is at the same level as the current directory.</td>
</tr>
<tr>
<td></td>
<td><code>unzip('mymfiles','../archives')</code></td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>zip</td>
</tr>
</tbody>
</table>
**Purpose**

Convert string to uppercase

**Syntax**

\[
\text{t} = \text{upper}('\text{str}') \\
\text{B} = \text{upper} (A)
\]

**Description**

\[
t = \text{upper} ('\text{str}') \text{ converts any lowercase characters in the string } \text{str} \text{ to the corresponding uppercase characters and leaves all other characters unchanged.}
\]

\[
\text{B} = \text{upper} (A) \text{ when } A \text{ is a cell array of strings, returns a cell array the same size as } A \text{ containing the result of applying upper to each string within } A.
\]

**Examples**

\[
\text{upper} ('\text{attention}!') \text{ is ATTENTION}!.
\]

**Remarks**

Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

**See Also**

lower
Purpose
Read contents 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-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('ftp://ftp.mathworks.com/pub/pentium/Moler_1.txt')

The file Moler_1.txt displays in the MATLAB Command Window.

Download Content from Local File
s = urlread('file:///c:/winnt/matlab.ini')
urlread

See Also    urlwrite
vander

2vander

Purpose
Vandermonde matrix

Syntax
A = vander(v)

Description
A = vander(v) returns the Vandermonde matrix whose columns are powers of
the vector v, that is, A(i,j) = v(i)^(n-j), where n = length(v).

Examples
vander(1:.5:3)

ans =

    1.0000    1.0000    1.0000    1.0000    1.0000
   5.0625    3.3750    2.2500    1.5000    1.0000
  16.0000    8.0000    4.0000    2.0000    1.0000
 39.0625   15.6250    6.2500    2.5000    1.0000
 81.0000   27.0000    9.0000    3.0000    1.0000

See Also
gallery
**Purpose**  
Variance

**Syntax**  
\( \text{var}(X) \)  
\( \text{var}(X,1) \)  
\( \text{var}(X,w) \)

**Description**  
\( \text{var}(X) \) returns the variance of \( X \) for vectors. For matrices, \( \text{var}(X) \) is a row vector containing the variance of each column of \( X \). \( \text{var}(X) \) normalizes by \( N-1 \) where \( N \) is the sequence length. This makes \( \text{var}(X) \) the best unbiased estimate of the variance if \( X \) is a sample from a normal distribution.

\( \text{var}(X,1) \) normalizes by \( N \) and produces the second moment of the sample about its mean.

\( \text{var}(X,w) \) computes the variance using the weight vector \( W \). The number of elements in \( W \) must equal the number of rows in \( X \) unless \( W = 1 \), which is treated as a short-cut for a vector of ones. The elements of \( W \) must be positive. \( \text{var} \) normalizes \( W \) by dividing each element in \( W \) by the sum of all its elements.

The variance is the square of the standard deviation (STD).

**See Also**  
corrcoef, cov, std
varargin, varargout

**Purpose**
Pass or return variable numbers of arguments

**Syntax**

```matlab
function varargout = foo(n)
function y = bar(varargin)
```

**Description**

function varargout = foo(n) returns a variable number of arguments from function foo.m.

function y = bar(varargin) accepts a variable number of arguments into function bar.m.

The varargin and varargout statements are used only inside a function M-file to contain the optional arguments to the function. Each must be declared as the last argument to a function, collecting all the inputs or outputs from that point onwards. In the declaration, varargin and varargout must be lowercase.

**Examples**
The function

```matlab
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

```matlab
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 ':'.

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.
varargin, varargout

See Also  nargin, nargout, nargchk, narginchk, 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

Display 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

```
ver
ver product
v = ver('product')
```

Description

The `ver` function 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.

The `ver product` function displays the MATLAB header information followed by the current version number for the specified 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 Systems 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 Systems Toolbox by typing

```
ver control
```

MATLAB returns

```
----------------------------------------------------------------
MATLAB Version 7.0.0.19220 (R14)
MATLAB License Number: %$#)^(%()$^
----------------------------------------------------------------
```

2-2458
Operating System: Microsoft Windows 2000 Version 5.0 (Build 2195: Service Pack 3)
Java VM Version: Java 1.4.2 with Sun Microsystems Inc. Java HotSpot(TM) Client VM

Control System Toolbox                  Version 6.0        (R14)

Return version information for the Control System Toolbox in a structure array, v.

```matlab
v = ver('control')
```

```
v =
    Name: 'Control System Toolbox'
    Version: '6.0'
    Release: '(R14)'
    Date: '19-Apr-2004'
```

**See Also**

help, hostid, license, version, whatsnew

Also, type `help info` at the Command Window prompt.
**Purpose**
Version control operations on PC platforms

**Graphical Interface**
As an alternative to the `verctrl` function, use Source Control in the Editor, Simulink, or Stateflow File menu.

**Syntax**

```matlab
fileChange = verctrl('command',{'filename1','filename2',....},handle)
verctrl('command',{ 'filename1','filename2',....}, handle)
fileChange = verctrl('command','file', handle)
verctrl('command','file', handle)
list = verctrl('all_systems')
```

**Description**
`fileChange=verctrl('command',{ 'filename1','filename2',....}, handle)` performs the version control specified by `command` on a single file or multiple files. Specify files with a cell array using the full pathnames for `filename`. On Windows, specify a Windows handle in the `handle` argument. These commands return a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk. Available values for `command` with this syntax are as follows:

<table>
<thead>
<tr>
<th>command Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'get'</td>
<td>Retrieves file(s) for viewing and compiling, but not editing. The file(s) will be tagged read-only. The list of files should contain either files or directories but not both.</td>
</tr>
<tr>
<td>'checkout'</td>
<td>Retrieves file(s) for editing.</td>
</tr>
<tr>
<td>'checkin'</td>
<td>Checks file(s) into the version control system, storing the changes and creating a new version.</td>
</tr>
<tr>
<td>'uncheckout'</td>
<td>Cancels a previous check-out operation and restores the contents of the selected file(s) to the precheckout version. All changes made to the file since the check-out are lost.</td>
</tr>
</tbody>
</table>
verctrl('command', {'filename1', 'filename2', ..., }, handle) performs the version control specified by 'command' on a single file or multiple files. Specify the files with a cell array using the full pathnames for 'filename'. On Windows, specify a Windows handle in the handle argument. Available values for 'command' with this syntax are as follows:

<table>
<thead>
<tr>
<th>command argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'add'</td>
<td>Adds file(s) into the version control system.</td>
</tr>
<tr>
<td>'history'</td>
<td>Displays the history of file(s).</td>
</tr>
<tr>
<td>'remove'</td>
<td>Removes file(s) from the version control system. It does not delete the file(s) from the local hard drive, only from the version control system.</td>
</tr>
</tbody>
</table>

fileChange = verctrl('command', 'file', handle) performs the version control specified by 'command' on a single file. Use the full pathname for 'file'. On Windows, specify a Windows handle in the handle argument. These commands return a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk. Available values for 'command' with this syntax are as follows:

<table>
<thead>
<tr>
<th>command argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>'properties'</td>
<td>Displays the properties of a file.</td>
</tr>
<tr>
<td>'isdiff'</td>
<td>Compares a file with the latest checked in version of the file in the version control system. Returns logical 1 to the workspace if the files are different and it returns logical 0 to the workspace if the files are identical.</td>
</tr>
</tbody>
</table>
verctrl

verctrl('command','file') performs the version control specified by 'command' on a single file. Use the full pathname for 'file'. Available values for 'command' with this syntax are as follows:

<table>
<thead>
<tr>
<th>command 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 version control system.</td>
</tr>
</tbody>
</table>

**Examples**

**Create a Windows Handle**
The verctrl function supports different version control commands on PC platforms. When you use verctrl on Windows, you must create a Windows handle. To create a Windows handle, enter the following commands in the MATLAB Command Window:

```matlab
>> import java.awt.*;
>> frame = Frame('Test frame');
>> frame.setVisible(1);
>> winhandle =
    com.mathworks.util.NativeJava.hWndFromComponent(frame)
```

**List Installed Source Control Systems**
Return a List in the Command Window of All Version Control Systems Installed in the Machine

```matlab
list = verctrl('all_systems')
list =
    'Microsoft Visual SourceSafe'
```

**Check Out a File**
Check out D:\file1.ext from the version control system. This command opens 'checkout' window and returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk.

```matlab
fileChange = verctrl('checkout',{'D:\file1.ext'}, 0)
```
Add Files
Add D:\file1.ext and D:\file2.ext to the version control system. This command opens 'add' window and returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk.

fileChange = verctrl('add',{'D:\file1.ext','D:\file2.ext'}, 0)

Display the Properties of a File
Display the properties of D:\file1.ext. This command opens 'properties' window and returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk.

fileChange = verctrl('properties','D:\file1.ext', 0)

See Also
checkin, checkout, undocheckout, cmopts
**version**

**Purpose**
Get MATLAB version number

**Graphical Interface**
As an alternative to the `version` function, select **About** from the **Help** menu in the MATLAB desktop.

**Syntax**

```
version
version -java
v = version
[v,d] = version
```

**Description**

`version` displays the MATLAB version number.

`version -java` displays the version of the Java VM (JVM) used by MATLAB.

`v = version` returns a string `v` containing the MATLAB version number.

`[v,d] = version` also returns a string `d` containing the date of the version.

**Remarks**
On the PC 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 26612 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**

```
[v,d]=version
```

```
v =
7.0.0.275711 (R14)

d =
Apr 16 2004
```

**See Also**
`ver`, `whatsnew`
Purpose  
Vertical concatenation

Syntax  
\[ C = \text{vertcat}(A_1, A_2, \ldots) \]

Description  
\( C = \text{vertcat}(A_1, A_2, \ldots) \) vertically concatenates matrices \( A_1, A_2, \) and so on. All matrices in the argument list must have the same number of columns. \text{vertcat} concatenates N-dimensional arrays along the first dimension. The remaining dimensions must match.

MATLAB calls \( C = \text{vertcat}(A_1, A_2, \ldots) \) for the syntax \( C = [A_1; A_2; \ldots] \) when any of \( A_1, A_2, \text{etc.} \) is an object.

Examples  
Create a 5-by-3 matrix, \( A \), and a 3-by-3 matrix, \( B \). Then vertically concatenate \( A \) and \( B \).

\[
A = \text{magic}(5); \quad \% \text{Create 5-by-3 matrix, } A \\
A(:, 4:5) = [] \quad \% \text{Create 5-by-3 matrix, } A
\]

\[
A = \\
\begin{array}{ccc}
17 & 24 & 1 \\
23 & 5 & 7 \\
4 & 6 & 13 \\
10 & 12 & 19 \\
11 & 18 & 25
\end{array}
\]

\[
B = \text{magic(3)} \times 100 \quad \% \text{Create 3-by-3 matrix, } B
\]

\[
B = \\
\begin{array}{ccc}
800 & 100 & 600 \\
300 & 500 & 700 \\
400 & 900 & 200
\end{array}
\]

\[
C = \text{vertcat}(A,B) \quad \% \text{Vertically concatenate } A \text{ and } B
\]

\[
C = 
\]
**vertcat**

17    24    1
23    5     7
4     6     13
10    12    19
11    18    25
800   100   600
300   500   700
400   900   200

**See Also**

horzcat, cat
Purpose

Viewpoint specification

Syntax

view(az,el)
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 \text{viewmtx}.

[az,el] = view returns the current azimuth and elevation.

\( T = \text{view} \) 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°.

```matlab
az = 180;
el = 90;
view(az, el);
```

**See Also**

viewmtx, axes, rotate3d
“Controlling the Camera Viewpoint” for related functions

Axes graphics object properties CameraPosition, CameraTarget, CameraViewAngle, Projection

Defining the View for more information on viewing concepts and techniques
Purpose

View transformation matrices

Syntax

\[
T = \text{viewmtx}(az, el) \\
T = \text{viewmtx}(az, el, \phi) \\
T = \text{viewmtx}(az, el, \phi, xc)
\]

Description

\text{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).

\[
T = \text{viewmtx}(az, el)
\]

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

\[
\text{view}(az, el) \\
T = \text{view}
\]

but does not change the current view.

\[
T = \text{viewmtx}(az, el, \phi)
\]

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 \(\text{view}(T)\). The 4-by-4 perspective transformation matrix transforms four-dimensional homogeneous vectors into unnormalized vectors of the form \((x, y, z, w)\), where \(w\) is not equal to 1. The \(x\) and \(y\)-components of the normalized vector \((x/w, y/w, z/w, 1)\) are the desired two-dimensional components (see example below).
T = 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.

```matlab
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

```matlab
x = [0 1 0 0 0 0 1 0 0 1 1 1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 0 0 1 1 1 1];
y = [0 0 1 1 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 1];
z = [0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1];
```
Transform the points in these vectors to the screen, then plot the object.

```matlab
A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A * x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:,1) = x2d(1,:);
y2(:,1) = x2d(2,:);
plot(x2,y2)
```
Use a perspective transformation with a 25 degree viewing angle:

```matlab
A = viewmtx(-37.5,30,25);
x4d = [.5 0 -3 1]';
x2d = A * x4d;
x2d = x2d(1:2)/x2d(4) % Normalize
x2d =
    0.1777
   -1.8858
```

Transform the cube vectors to the screen and plot the object:

```matlab
A = viewmtx(-37.5,30,25);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A * x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)/x2d(4,:);
y2(:) = x2d(2,:)/x2d(4,:);
plot(x2,y2)
```
See Also

view, hgtransform

"Controlling the Camera Viewpoint" for related functions

Defining the View for more information on viewing concepts and techniques
**volumebounds**

**Purpose**

Return coordinate and color limits for volume data

**Syntax**

```matlab
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:

```
[xmin xmax ymin ymax zmin zmax cmin 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:

```
[xmin xmax ymin ymax zmin zmax]
```

`lims = volumebounds(V), lims = volumebounds(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(V)`.

**Examples**

This example uses `volumebounds` to set the axis and color limits for an isosurface generated by the `flow` function.

```matlab
[x y z v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
daspect([1 1 1])
isocolors(x,y,z,flipdim(v,2),p)
shading interp
axis(volumebounds(x,y,z,v))
view(3)
camlight
lighting phong
```
See Also  
isosurface, streamslice

“Volume Visualization” for related functions
Voronoi diagram

Consider a set of coplanar points. For each point in the set, you can draw a boundary enclosing all the intermediate points lying closer to than to other points in the set. 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.

voronoi(x,y) plots the bounded cells of the Voronoi diagram for the points x,y. Cells that contain a point at infinity are unbounded and are not plotted.

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.

Note For the topology of the Voronoi diagram, i.e., the vertices for each Voronoi cell, use voronoin.
[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.

```matlab
xlim([min(x) max(x)])
ylim([min(y) max(y)])
```

**Example 3.** This code uses `voronoin` and `patch` to fill the bounded cells of the same Voronoi diagram with color.
voronoi

```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.
        patch(v(c{i},1),v(c{i},2),i); % use color i.
    end
end
axis equal
```

Algorithm
If you supply no triangulation TRI, the voronoi function performs a Delaunay triangulation of the data that uses Qhull [2]. 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

N-dimensional Voronoi diagram

.Syntax

```matlab
[V,C] = voronoin(X)
[V,C] = voronoin(X,options)
```

.Description

`[V,C] = voronoin(X)` returns Voronoi vertices `V` and the Voronoi cells `C` of the Voronoi diagram of `X`. `V` is a `numv`-by-`n` array of the `numv` Voronoi vertices in `n`-dimensional space, each row corresponds to a Voronoi vertex. `C` is a vector cell array where each element contains the indices into `V` of the vertices of the corresponding Voronoi cell. `X` is an `m`-by-`n` array, representing `m` `n`-dimensional points, where `n > 1` and `m >= 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.

`voronoin` uses Qhull.

`[V,C] = voronoin(X,options)` specifies a cell array of strings `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 `options` is `[]`, the default options are used. If `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 `convhulln` to compute the vertices of the facets that make up the Voronoi cell. Then use `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 documentation.

.Examples

Let

```matlab
x = [ 0.5  0
     0  0.5
    -0.5 -0.5
    -0.2 -0.1
    -0.1  0.1
```
then

\[ [V,C] = \text{voronoin}(x) \]

\[ V = \begin{bmatrix}
  \text{Inf} & \text{Inf} \\
  0.3833 & 0.3833 \\
  0.7000 & -1.6500 \\
  0.2875 & 0.0000 \\
  -0.0000 & 0.2875 \\
  -0.0000 & -0.0000 \\
  -0.0500 & -0.5250 \\
  -0.0500 & -0.0500 \\
  -1.7500 & 0.7500 \\
  -1.4500 & 0.6500
\end{bmatrix} \]

\[ C = \begin{bmatrix}
[1 \times 4 \text{ double}] \\
[1 \times 5 \text{ double}] \\
[1 \times 4 \text{ double}] \\
[1 \times 4 \text{ double}] \\
[1 \times 4 \text{ double}] \\
[1 \times 5 \text{ double}] \\
[1 \times 4 \text{ double}]
\end{bmatrix} \]

Use a for loop to see the contents of the cell array \( C \).

\[ \text{for } i=1: \text{length}(C), \text{disp}(C(i)), \text{end} \]

\[ \begin{bmatrix}
4 & 2 & 1 & 3 \\
10 & 5 & 2 & 1 & 9 \\
9 & 1 & 3 & 7 \\
10 & 8 & 7 & 9 \\
10 & 5 & 6 & 8 \\
8 & 6 & 4 & 3 & 7 \\
6 & 4 & 2 & 5
\end{bmatrix} \]
2wait

Purpose
Wait until a 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**
Display waitbar

**Syntax**

```matlab
h = waitbar(x,'title')
waitbar(x,'title','CreateCancelButton','button_callback')
waitbar(...,property_name,property_value,...)
waitbar(x)
waitbar(x,h)
waitbar(x,h,'updated title')
```

**Description**
A waitbar shows what percentage of a calculation is complete, as the calculation proceeds.

```matlab
h = waitbar(x,'title') displays a waitbar of fractional length x. The handle to the waitbar figure is returned in h. x must be between 0 and 1.

waitbar(x,'title','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 corresponding waitbar figure properties.

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.
```
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” 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`

“Interactive User Input” for related functions
**Purpose**
Wait for key or mouse button press

**Syntax**
k = waitforbuttonpress

**Description**
k = waitforbuttonpress blocks the caller’s execution stream until the function detects that the user has pressed a mouse button or a key while the figure window is active. The function returns

- 0 if it detects a mouse button press
- 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 press')
else
    disp('Key press')
end
```

**See Also**
dragrect, ginput, rbbox, waitfor

“Developing User Interfaces” for related functions
warndlg

Purpose
Display warning dialog box

Syntax
h = warndlg('warningstring','dlgname')

Description
warndlg displays a dialog box named 'Warning Dialog' containing the string 'This is the default warning string.' The warning dialog box disappears after you press the OK button.

warndlg('warningstring') displays a dialog box with the title 'Warning Dialog' containing the string specified by warningstring.

warndlg('warningstring','dlgname') displays a dialog box with the title dlgname that contains the string warningstring.

h = warndlg(...) returns the handle of the dialog box.

Examples
The statement
    warndlg('Pressing OK will clear memory','!! Warning !!')

displays this dialog box:

See Also
warning, dialog, errordlg, helpdlg, msgbox

“Predefined Dialog Boxes” for related functions
**Purpose**
Display 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.

```
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 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 “Control Statements” in the MATLAB documentation for more information.

Output s is a structure array that indicates the current state of the selected warnings. The structure has the fields identifier and state. See “Output from Control Statements” in the MATLAB documentation for more.

`s = warning(state, mode)` is a warning control statement that enables you to enter debug mode, 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 ‘debug’, ‘backtrace’, or ‘verbose’. See “Debug, Backtrace, and Verbose” in the MATLAB 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
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.
```

To enter debug mode whenever a parameterNotSymmetric warning is invoked in a component called Control, first turn off all warnings and enable only this one type of warning using its message identifier. Then turn on debug mode for all enabled warnings. When you run your program, MATLAB will stop in debug mode just before this warning is executed. You will see the debug prompt (K>>) displayed:

```matlab
warning off all
warning on Control:parameterNotSymmetric
warning on debug
```

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, lasterr, errordlg, dbstop, disp, sprintf
```
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:size(Z,1)` and `y = 1: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 `length(x) = n`, `length(y) = m`, and `[m,n] = 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_handle,...)` 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.

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`. 

Remarks

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`. 

Syntex

- `waterfall(Z)`
- `waterfall(X,Y,Z)`
- `waterfall(...,C)`
- `waterfall(axes_handle,...)`
- `h = waterfall(...)`
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’. 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.
**wavinfo**

**Purpose**
Return information about Microsoft WAVE (.wav) sound file

**Syntax**

```
[m d] = wavinfo(filename)
```

**Description**

```
[m d] = wavinfo(filename) returns information about the contents of the
WAVE sound file specified by the string filename.

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 a PC-based audio output device

**Syntax**

```
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'` (default value): 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'`: 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 `Fs`. Load and play the gong and the chirp audio signals.
Change the names of these signals in between `load` commands and play them sequentially using the `'sync'` option for `wavplay`.

```matlab
load chirp;
y1 = y; Fs1 = Fs;
load gong;
wavplay(y1,Fs1,'sync') % The chirp signal finishes before the
wavplay(y,Fs)       % gong signal begins playing.
```

See Also

`wavrecord`
### Purpose
Read Microsoft WAVE (.wav) sound file

### Graphical Interface
As an alternative to audioread, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

### Syntax
- `y = wavread('filename')`
- `[y,Fs,bits] = wavread('filename')`
- `[...,N] = wavread('filename',N)`
- `[...,N1 N2] = wavread('filename',[N1 N2])`
- `[...,‘size’] = wavread('filename’,‘size’)`

### Description
`wavread` supports multichannel data, with up to 32 bits per sample, and supports reading 24- and 32-bit .wav files.

- `y = wavread('filename')` loads a WAVE file specified by the string `filename`, returning the sampled data in `y`. The `.wav` extension is appended if no extension is given. Amplitude values are in the range `[-1,+1]`.

- `[y,Fs,bits] = wavread('filename')` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.

- `[...,N] = wavread('filename',N)` returns only the first `N` samples from each channel in the file.

- `[...,N1 N2] = wavread('filename',[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

- `siz = wavread('filename’,‘size’)` returns the size of the audio data contained in the file in place of the actual audio data, returning the vector `siz = [samples channels]`.

### See Also
- audioread
- wavwrite
wavrecord

Purpose

Record sound using a 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 \( '\text{dtype}' \) to record the sound. The string \( '\text{dtype}' \) 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, 22500, 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 \text{wavplay}. Speak into your audio device (or produce your audio signal) while the \text{wavrecord} command runs.

\[ Fs = 11025; \]
\[ y = \text{wavrecord}(5*Fs,Fs,'\text{int16}'); \]
\[ \text{wavplay}(y,Fs); \]

See Also

\text{wavplay}
**Purpose**
Write a 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 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
**web**

**Purpose**
Open Web site or file in Web browser or Help browser

**Syntax**
web
web url
web url -new
web url -notoolbar
web url -noaddressbox
web url -helpbrowser
web url -browser
web(...)

\[
\text{stat} = \text{web}('url', '\text{-browser}')
\]

\[
[\text{stat}, \text{h1}] = \text{web}
\]

\[
[\text{stat}, \text{h1}, \text{url}] = \text{web}
\]

**Description**
web opens an empty MATLAB Web browser. The MATLAB Web browser includes an address field where you can enter a URL (Uniform Resource Locator), 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 in the MATLAB Web browser. If any MATLAB Web browsers are already open, displays the page in the browser that last had focus.

web url -new displays the specified url in a new MATLAB Web browser.

web url -notoolbar displays the specified 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 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 displays in the browser that last had focus, regardless of its address field status.

web url -helpbrowser displays the specified 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 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, 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, for the last active browser, and returns its current URL to url.
Examples


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 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, matlab:
weekday

Purpose
Display current day of the week

Syntax
[N, S] = weekday(D)
[N, S] = weekday(D, form)
[N, S] = weekday(D, locale)
[N, S] = weekday(D, form, locale)

Description
[N, S] = weekday(D) returns the day of the week in numeric (N) and string (S) form for a given serial date number or date string D. Input argument D can represent more than one date in an array of serial date numbers or a cell array of date strings.

[N, S] = weekday(D, form) returns the day of the week in numeric (N) and string (S) form, where the content of S depends on the form argument. If form is 'long', then S contains the full name of the weekday (e.g., Tuesday). If form is 'short', then S contains an abbreviated name (e.g., Tues) from this table.

The days of the week are assigned these numbers and abbreviations.

<table>
<thead>
<tr>
<th>N</th>
<th>S (short)</th>
<th>S (long)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sun</td>
<td>Sunday</td>
</tr>
<tr>
<td>2</td>
<td>Mon</td>
<td>Monday</td>
</tr>
<tr>
<td>3</td>
<td>Tue</td>
<td>Tuesday</td>
</tr>
<tr>
<td>4</td>
<td>Wed</td>
<td>Wednesday</td>
</tr>
<tr>
<td>5</td>
<td>Thu</td>
<td>Thursday</td>
</tr>
<tr>
<td>6</td>
<td>Fri</td>
<td>Friday</td>
</tr>
<tr>
<td>7</td>
<td>Sat</td>
<td>Saturday</td>
</tr>
</tbody>
</table>

[N, S] = weekday(D, locale) returns the day of the week in numeric (N) and string (S) form, where the format of the output depends on the locale argument. If locale is 'local', then weekday uses local format for its output. If locale is 'en_US', then weekday uses US English.
weekday

[N, S] = weekday(D, form, locale) returns the day of the week using the formats described above for form and locale.

**Examples**

Either

\[
[n, s] = \text{weekday}(728647)
\]

or

\[
[n, s] = \text{weekday('19-Dec-1994')}
\]

returns \( n = 2 \) and \( s = \text{Mon} \).

**See Also**

datenum, datevec, eomday
**What**

**Purpose**
List MATLAB specific 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**

```
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>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>Path to directory</td>
</tr>
<tr>
<td>m</td>
<td>Cell array of M-file names</td>
</tr>
<tr>
<td>mat</td>
<td>Cell array of MAT-file names</td>
</tr>
<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/audio:

```matlab
what audio
```

M-files in directory matlabroot/toolbox/matlab/audio

```matlab
% Contents
auread          soundsc
audiodevinfo    auwrite          wavplay
audioplayer     lin2mu           wavread
audioplayerreg  mu2lin           wavrecord
audiorecorder   prefspanel       wavwrite
audiorecorderreg saxis
audiouniquename sound
```

MAT-files in directory matlabroot/toolbox/matlab/audio

```matlab
chirp            handel          splat
gong            laughter        train
```

Obtain a structure array containing the MATLAB filenames in toolbox/matlab/general.

```matlab
s = what('general')
```

```matlab
s =
    path: 'matlabroot:\toolbox\matlab\general'
    m: {104x1 cell}
    mat: {0x1 cell}
    mex: {5x1 cell}
    mdl: {0x1 cell}
    p: {'helpwin.p'}
    classes: {'char'}
```

See Also

dir, exist, lookfor, mfilename, path, which, who
whatsnew

**Purpose**
Display 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 known problems, all organized by product.

**See Also**
help, lookfor, path, version, which
**Purpose**
Locate functions and files

**Graphical Interface**
As an alternative to the `which` function, use the Current Directory browser.

**Syntax**

- `which fun`
- `which classname/fun`
- `which private/fun`
- `which classname/private/fun`
- `which fun1 in fun2`
- `which fun(a,b,c,...)`
- `which file.ext`
- `which fun -all`
- `s = which('fun',...)`

**Description**

`which fun` displays the full pathname for the argument `fun`. If `fun` is a
- MATLAB function or Simulink model in an M, P, or MDL file on the MATLAB path, then `which` displays the full pathname for the corresponding file
- Workspace variable, then `which` displays a message identifying `fun` as a variable
- Method in a loaded Java class, then `which` displays the package, class, and method name for that method

If `fun` is an overloaded function or method, then `which fun` returns only the pathname of the first function or method found.

- `which classname/fun` displays the full pathname for the M-file defining the `fun` method in MATLAB class, `classname`. For example, `which serial/fopen` displays the path for `fopen.m` in the MATLAB class directory, `@serial`.

- `which private/fun` limits the search to private functions. For example, `which private/orthog` displays the path for `orthog.m` in the `/private` subdirectory of `toolbox/matlab/elmat`.

- `which classname/private/fun` limits the search to private methods defined by the MATLAB class, `classname`. For example, `which dfilt/private/todtf` displays the path for `todtf.m` in the `private` directory of the `dfilt` class.
which fun1 in fun2 displays the pathname to function fun1 in the context of
the M-file fun2. You can use this form to determine whether a subfunction or
private version of fun1 is called from fun2, rather than 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:

```matlab
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:

```matlab
s = which('private/stradd','-all');
whos s
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>3x1</td>
<td>562</td>
<td>cell array</td>
</tr>
</tbody>
</table>

Grand total is 146 elements using 562 bytes

**See Also**
dir, doc, exist, lookfor, mfilename, path, type, what, who
**while**

**Purpose**
Repeat statements an indefinite number of times

**Syntax**

```matlab
while expression
    statements
end
```

**Description**
while repeats statements an indefinite number of times. The statements are executed while the real part of expression has all nonzero elements. 

expression is usually of the form

```matlab
expression rel_op expression
```

where rel_op is ==, <, <=, >, or >=.

The scope of a while statement is always terminated with a matching end.

**Arguments**

expression

expression is a MATLAB expression, usually consisting 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.

```matlab
(count < limit) & ((height - offset) >= 0)
```

statements

statements is one or more MATLAB statements to be executed only while the expression is true or nonzero.

**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, below.

2-2512
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 ~= 0) & (a/b > 18.5)
if exist('myfun.m') & (myfun(x) >= y)
if iscell(A) & all(cellfun('isreal', A))

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.

eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
es = eps*2
Example 2 - Nonscalar Expression

Given matrices $A$ and $B$,

$$A = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 \\ 3 & 4 \end{bmatrix}$$

<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>
<tr>
<td>$A &lt; (B + 1)$</td>
<td>true</td>
<td>Every element of $A$ is less than that same element of $B$</td>
</tr>
<tr>
<td>$A &amp; B$</td>
<td>false</td>
<td>$A(1,2) &amp; B(1,2)$ is false.</td>
</tr>
<tr>
<td>$B &lt; 5$</td>
<td>true</td>
<td>Every element of $B$ is less than 5.</td>
</tr>
</tbody>
</table>

See Also

end, for, break, continue, return, all, any, if, switch
Purpose
Change axes background color

Syntax
whitebg
whitebg(h)
whitebg(ColorSpec)
whitebg(h,ColorSpec)

Description
whitebg complements the colors in the current figure.
whitebg(h) complements colors in all figures specified in the vector h.
whitebg(ColorSpec) and whitebg(h,ColorSpec) change the color of the axes,
which are children of the figure, to the color specified by ColorSpec.

Remarks
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 0.5 0.6])
Set the background color to blue.
whitebg('blue')

See Also
ColorSpec
The figure graphics object property InvertHardCopy
“Color Operations” for related functions
who, whos

**Purpose** List variables in the workspace

**Graphical Interface** As an alternative to whos, use the Workspace browser.

**Syntax** Each of these syntaxes apply to both who and whos:

```plaintext
who
who(variable_list)
who(variable_list, qualifiers)
s = who(variable_list, qualifiers)
who variable_list qualifiers
```

**Description** who lists the variables currently in the workspace.

whos lists the current variables and their sizes and types. It also reports the totals for sizes.

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>
</tbody>
</table>
who, whos

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:

- name: variable name
- size: variable size
- bytes: number of bytes allocated for the array
- class: class of variable

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

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.

Examples

Show variable names starting with the letter a:

```matlab
who a*
```

Show variables stored in MAT-file mydata.mat that start with java and end with Array. Also show their dimensions and class name:

```matlab
whos -file mydata -regexp \<java.*Array\>
```
who, whos

<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>javaDb1Array</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>

Get variable names that start with uppercase or lowercase X, are followed by zero or more other characters, and end with at least two digits. Assign the output to cell array xvars:

```matlab
xvars = who('-regexp', '[Xx].*\d{2,}$')
xvars =
    'X_JohnDavis_5May03'
    'Xtest_120'
    'xArray5by12'
    'x_times_3pt82'
```

**See Also**
assignin, clear, computer, dir, evalin, exist, inmem, load, save, what, workspace
**Purpose**

Wilkinson's eigenvalue test matrix

**Syntax**

\[ W = \text{wilkinson}(n) \]

**Description**

\( W = \text{wilkinson}(n) \) 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**

\[ \text{wilkinson}(7) \]

\[
\begin{array}{cccccccc}
3 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 2 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 2 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 3 \\
\end{array}
\]

The most frequently used case is \( \text{wilkinson}(21) \). 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 only)

**Syntax**  
`winopen('filename')`

**Description**  
`winopen('filename')` opens `filename` in the appropriate Microsoft Windows application. 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:

```
winopen('thesis.doc')
```

Open `myresults.html` in your system’s default Web browser:

```
winopen('D:/myfiles/myresults.html')
```

**See Also**  
dos, open, web
**Purpose**

Get item from Microsoft Windows registry

**Syntax**

```matlab
value = winqueryreg('name', 'rootkey', 'subkey')
value = winqueryreg('rootkey', 'subkey', 'valname')
value = winqueryreg('rootkey', 'subkey')
```

**Description**

`value = winqueryreg('name', 'rootkey', 'subkey')` returns the key names in `rootkey\subkey` in a cell array of strings. The first argument is the literal quoted string, `name`.

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', 'valname')` returns the value for key `valname` in `rootkey\subkey`.

`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 `mwsampctrl.2`:

```matlab
winqueryreg 'HKEY_CLASSES_ROOT' 'mwsamp.mwsampctrl.2\clsid'
```

```matlab
ans =
{5771A80A-2294-4C4C-A75B-157DCDDD3653}
```
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

See Also
Purpose
Determine if file contains Lotus WK1 worksheet

Syntax
[extens, type] = wk1finfo('filename')

Description
[extens, type] = wk1finfo('filename') returns the string 'WK1' in extens, and 'Lotus 123 Spreadsheet' in type if the file filename contains a readable Lotus worksheet.

Examples
This example returns information on spreadsheet file matA.wk1:

```
[extens, type] = wk1finfo('matA.wk1')
```

```
extens =
    WK1
type =
    Lotus 123 Spreadsheet
```

See Also
wk1read, wk1write, csvread, csvwrite


**wk1read**

**Purpose**
Read Lotus123 spreadsheet file (.wk1)

**Syntax**

\[
\begin{align*}
M &= \text{wk1read}(\text{filename}) \\
M &= \text{wk1read}(\text{filename},r,c) \\
M &= \text{wk1read}(\text{filename},r,c,\text{range})
\end{align*}
\]

**Description**

\[
M = \text{wk1read}(\text{filename}) \text{ reads a Lotus123 WK1 spreadsheet file into the matrix } M.
\]

\[
M = \text{wk1read}(\text{filename},r,c) \text{ starts reading at the row-column cell offset specified by } (r,c). \text{ } r \text{ and } c \text{ are zero based so that } r=0, c=0 \text{ specifies the first value in the file.}
\]

\[
M = \text{wk1read}(\text{filename},r,c,\text{range}) \text{ reads the range of values specified by the parameter range, where range can be}
\]

- A four-element vector specifying the cell range in the format
  \[
  \begin{bmatrix}
  \text{upper_left_row} & \text{upper_left_col} & \text{lower_right_row} & \text{lower_right_col}
  \end{bmatrix}
  \]

- A cell range specified as a string, for example, 'A1...C5'

- A named range specified as a string, for example, 'Sales'

**Examples**

Create a 8-by-8 matrix A and export it to Lotus spreadsheet matA.wk1:

\[
A = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\
31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\
41 & 42 & 43 & 44 & 45 & 46 & 47 & 48
\end{bmatrix}
\]
To read in a limited block of the spreadsheet data, specify the upper left row and column of the block using zero-based indexing:

\[
M = \text{wk1read}('matA.wk1', 3, 2)
\]

\[
M =
\begin{bmatrix}
33 & 34 & 35 & 36 & 37 & 38 \\
43 & 44 & 45 & 46 & 47 & 48 \\
53 & 54 & 55 & 56 & 57 & 58 \\
63 & 64 & 65 & 66 & 67 & 68 \\
73 & 74 & 75 & 76 & 77 & 78
\end{bmatrix}
\]

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:

\[
M = \text{wk1read}('matA.wk1', 3, 2, [4 \ 3 \ 6 \ 6])
\]

\[
M =
\begin{bmatrix}
33 & 34 & 35 & 36 \\
43 & 44 & 45 & 46 \\
53 & 54 & 55 & 56
\end{bmatrix}
\]

See Also

wk1write
**wk1write**

**Purpose**
Write a matrix to a Lotus123 WK1 spreadsheet file

**Syntax**

\[
\text{wk1write(filename, M)} \\
\text{wk1write(filename, M, r, c)}
\]

**Description**

\(\text{wk1write(filename, M)}\) writes the matrix \(M\) into a Lotus123 WK1 spreadsheet file named \(filename\).

\(\text{wk1write(filename, M, r, c)}\) 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:

\[
A = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
11 & 12 & 13 & 14 & 15 \\
21 & 22 & 23 & 24 & 25 \\
31 & 32 & 33 & 34 & 35
\end{bmatrix}
\]

\[
M = \text{wk1write('matA.wk1', A, 2, 3)}
\]

\[
M = \text{wk1read('matA.wk1')}
\]

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 2 & 3 & 4 & 5 \\
0 & 0 & 0 & 11 & 12 & 13 & 14 & 15 \\
0 & 0 & 0 & 21 & 22 & 23 & 24 & 25
\end{bmatrix}
\]
See Also

wk1read, dlmwrite, dlmread, csvwrite, csvread
workspace

Purpose
Display the Workspace browser, a tool for managing the workspace

Graphical Interface
As an alternative to the workspace function, select Workspace from the Desktop menu 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.

To see and edit a graphical representation of a variable, double-click the variable in the Workspace browser. The variable is displayed in the Array Editor, where you can edit it.

See Also
who
xlabel, ylabel, zlabel

2xlabel, ylabel, zlabel

Purpose
Label the x-, y-, and z-axis

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.
**xlabel, ylabel, zlabel**

**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.

```matlab
xlabel({'first line';'second line'})
```

**See Also**

text, title

“Annotating Plots” for related functions

Adding Axis Labels to Graphs for more information about labeling axes
**Purpose**
Set or query axis limits

**Syntax**
Note that the syntax for each of these three functions is the same; only the \texttt{xlim} function is used for simplicity. Each operates on the respective \(x\)-, \(y\)-, or \(z\)-axis.

\texttt{xlim}

\texttt{xlim([xmin xmax])}
\texttt{xlim('mode')}
\texttt{xlim('auto')}
\texttt{xlim('manual')}
\texttt{xlim(axes_handle,...)}

**Description**
\texttt{xlim} with no arguments returns the respective limits of the current axes.

\texttt{xlim([xmin xmax])} sets the axis limits in the current axes to the specified values.

\texttt{xlim('mode')} returns the current value of the axis limits mode, which can be either \texttt{auto} (the default) or \texttt{manual}.

\texttt{xlim('auto')} sets the axis limit mode to \texttt{auto}.

\texttt{xlim('manual')} sets the respective axis limit mode to \texttt{manual}.

\texttt{xlim(axes_handle,...)} performs the set or query on the axes identified by the first argument, \texttt{axes_handle}. When you do not specify an \texttt{axes_handle}, these functions operate on the current axes.

**Remarks**
\texttt{xlim}, \texttt{ylim}, and \texttt{zlim} set or query values of the axes object \texttt{XLim}, \texttt{YLim}, \texttt{ZLim}, and \texttt{XLimMode}, \texttt{YLimMode}, \texttt{ZLimMode} properties.

When the axis limit modes are \texttt{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 \texttt{manual}. Note that high-level plotting functions like \texttt{plot} and \texttt{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 \texttt{hold} command.
**xlim, ylim, zlim**

**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])
```

**See Also**

axis

The axes properties `XLim`, `YLim`, `ZLIm`

“Setting the Aspect Ratio and Axis Limits” for related functions

Understanding Axes Aspect Ratio for more information on how axis limits affect the axes
Purpose
Determine if file contains Microsoft Excel (.xls) spreadsheet

Syntax
\[
\text{type} = \text{xlsfinfo('filename')} \\
\text{[type, sheets]} = \text{xlsfinfo('filename')} \\
\text{xlsfinfo filename}
\]

Description
\[\text{type} = \text{xlsfinfo('filename')} \] returns the string 'Microsoft Excel Spreadsheet' in the type output if the file specified by filename can be read by the MATLAB function xlsread. Otherwise, type is the empty string.

\[\text{[type, sheets]} = \text{xlsfinfo('filename')} \] returns in the sheets output a cell array of strings containing the names of each spreadsheet in the file.

xlsfinfo filename is the command format for xlsfinfo.

Examples
When filename is an Excel spreadsheet,
\[
\text{[type, sheets]} = \text{xlsfinfo('myaccount.xls')}
\]
\[
\text{type} =
\text{Microsoft Excel Spreadsheet}
\]
\[
\text{sheets} =
\text{'Sheet1'  'Income'  'Expenses'}
\]

See Also
xlsread, xlswrite
**xlsread**

**Purpose**
Read Microsoft Excel spreadsheet file (.xls)

**Syntax**

\[
N = \text{xlsread('filename')}
\]
\[
N = \text{xlsread('filename', -1)}
\]
\[
N = \text{xlsread('filename', sheet)}
\]
\[
N = \text{xlsread('filename', 'range')}
\]
\[
N = \text{xlsread('filename', sheet, 'range')}
\]
\[
N = \text{xlsread('filename', sheet, 'range', 'basic')}
\]
\[
[N, T] = \text{xlsread('filename', ...)}
\]
\[
[N, T, rawdata] = \text{xlsread('filename', ...)}
\]

**Description**

\[N = \text{xlsread('filename')}\] returns numeric data in double array \(N\) from the first sheet in the Microsoft Excel spreadsheet file named \(filename\). The \text{xlsread} ignores leading rows or columns of text. However, if a cell not in a leading row or column is empty or contains text, \text{xlsread} puts a NaN in its place in the return array, \(N\).

\[N = \text{xlsread('filename', -1)}\] 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” below.)

\[N = \text{xlsread('filename', sheet)}\] reads the specified worksheet, where \(sheet\) is either a positive, double scalar value or a quoted string containing the sheet name. To determine the names of the sheets in a spreadsheet file, use \text{xlsfinfo}.

\[N = \text{xlsread('filename', 'range')}\] reads data from a specific rectangular region of the default worksheet (Sheet1). Specify \texttt{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” below.)
N = xlsread('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,” below.)

N = xlsread('filename', sheet, 'range', 'basic') imports data from the spreadsheet in basic import mode. sheet must be a quoted string and is case sensitive. range is ignored and can be set to the empty string ('').

[N, T] = xlsread('filename', ...) returns numeric data in array N and text data in cell array T. All cells in T that correspond to numeric data contain the empty string.

If T 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 T output as 'ActiveX VT_ERROR: '

[N, T, rawdata] = xlsread('filename', ...) returns numeric and text data in N and T, and unprocessed cell content in cell array rawdata, which contains both numeric and text data. (See “COM Server Requirements” 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

COM Server Requirements

The following four syntax formats are supported only on computer systems capable of starting Excel as a COM server from MATLAB.

N = xlsread('filename', -1)
N = xlsread('filename', 'range')
N = xlsread('filename', sheet, 'range')
[N, T, rawdata] = xlsread('filename', ...)
**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
3 8
4 9
5 text
```

`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'

See Also
xlswrite, xlsinfo, wk1read, textread
xlswrite

**Purpose**
Write Microsoft Excel spreadsheet file (.xls)

**Syntax**

```matlab
xlswrite('filename', M)
xlswrite('filename', M, sheet)
xlswrite('filename', M, '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 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.

\[ \text{xlswrite('filename', } M, \text{ sheet, 'range'}) \text{ writes matrix } M \text{ to a rectangular region specified by range in worksheet sheet of the file filename. See the previous two syntax formats for further explanation of the sheet and range inputs.} \]

\[ \text{status} = \text{xlswrite('filename', ...)} \text{ returns the completion status of the write operation in status. If the write completed successfully, status is equal to 1 (or true). Otherwise, status is 0 (or false). Unless you specify an output for xlswrite, no status is displayed in the Command Window.} \]

\[ \text{[status, message] = xlswrite('filename', ...)} \text{ returns any warning or error message generated by the write operation in the MATLAB structure message. The message structure has two fields:} \]

- \text{message} — String containing the text of the warning or error message
- \text{identifier} — String containing the message identifier for the warning or error

\[ \text{xlswrite filename } M \text{ sheet range is the command format for xlswrite, showing its usage with all input arguments specified. When using this format, you must specify sheet as a string (for example, Income or Sheet4). If the sheet name contains space characters, then quotation marks are required around the string (for example, ‘Income 2002’).} \]

**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:

\[ \text{xlswrite('testdata', [12.7 5.02 -98 63.9 0 -.2 56])} \]

**Example 2 — Writing Mixed Data to a Specific Worksheet**

This example writes the following mixed text and numeric data to the file tempdata.xls:

\[ d = \{'Time', 'Temp'; 12 98; 13 99; 14 97} ; \]
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 mmsg] = 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**
```plaintext```
DOMnode = xmlread(filename)
```

**Description**
`DOMnode = xmlread(filename)` reads a URL or filename and returns a Document Object Model node representing the parsed document.

**Remarks**

**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.

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,docNode);

See Also
xmlread, xslt
Purpose

Exclusive or

Syntax

\[ C = \text{xor}(A, B) \]

Description

\[ C = \text{xor}(A, B) \] performs an exclusive OR operation on the corresponding elements of arrays \( A \) and \( B \). The resulting element \( C(i,j,...) \) is logical true (1) if \( A(i,j,...) \) or \( B(i,j,...) \), but not both, is nonzero.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>0</td>
</tr>
<tr>
<td>Zero</td>
<td>Nonzero</td>
<td>1</td>
</tr>
<tr>
<td>Nonzero</td>
<td>Zero</td>
<td>1</td>
</tr>
<tr>
<td>Nonzero</td>
<td>Nonzero</td>
<td>0</td>
</tr>
</tbody>
</table>

Examples

Given \( A = [0 \ 0 \ \pi \ \text{eps}] \) and \( B = [0 \ -2.4 \ 0 \ 1] \), then

\[
C = \text{xor}(A,B)
\]

\[
C = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}
\]

To see where either \( A \) or \( B \) has a nonzero element and the other matrix does not,

\[ \text{spy}(\text{xor}(A,B)) \]

See Also

all, any, find, Elementwise Logical Operators, Short-Circuit Logical Operators
**Purpose**
Transform XML document using XSLT engine

**Syntax**

```
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/.

**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.

```
xslt info.xml info.xsl info.html -web
```

**See Also**
xmlread, xmlwrite
Purpose
Create an array of all zeros

Syntax
B = zeros(n)
B = zeros(m,n)
B = zeros([m n])
B = zeros(d1,d2,d3...)
B = zeros([d1 d2 d3...])
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(d1,d2,d3...) or B = zeros([d1 d2 d3...]) returns an array of zeros with dimensions d1-by-d2-by-d3-by-... .

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', or 'uint32'.

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

x = zeros(1,n);
for i = 1:n, x(i) = i; end
zeros

See Also  
  eye, ones, rand, randn
Purpose

Create compressed version of files in zip format

Syntax

zip('zipfilename','files')
zip('zipfilename','directory')
zip(...,'rootdirectory')

Description

zip('zipfilename','files') creates a zip file named zipfilename from the file named files. For multiple files, make files a cell array of strings. Paths for zipfilename and files are relative to the current directory. Zip files are often used for archiving or for minimizing file transmission time.

zip('zipfilename','directory') creates a zip file named zipfilename consisting of the specified directory and all the files in it. The paths for zipfilename and directory are relative to the current directory.

zip('zipfilename','source','rootdirectory') allows the path specified for source to be relative to 'rootdirectory' rather than to the current directory. Note that source cannot be an absolute path.

Examples

Zipping 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.

zip('d:/mymfiles/viewlet.zip','$matlabroot/demos/guide.viewlet')

Zip 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.

zip('viewlets.zip',{'guide.viewlet','import.viewlet'})

Zipping a Directory

Zip the directory D:/mymfiles and its contents to the zip file mymfiles in the directory one level up from the current directory.

zip('./mymfiles','D:/mymfiles')

Zip the files thesis.doc and defense.ppt, which are located in d:/PhD, to the zip file thesis.zip in the current directory.

zip('thesis.zip',{'thesis.doc','defense.ppt'},'d:/PhD')
zip

See Also unzip
Purpose
Zoom in and out on a 2-D plot

Syntax
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig, option)

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.

- For a single-button mouse, zoom in by pressing the mouse button and zoom out by simultaneously pressing Shift and the mouse button.
- For a two- or three-button mouse, zoom in by pressing the left mouse button and zoom out by pressing the right mouse button.

Clicking and dragging over an axes when interactive zooming is enabled draws a rubberband box. When the mouse button is released, 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.

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.

zoom xon and zoom yon set zoom on for the x- and y-axis, respectively.
zoom(factor) zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by 1/factor.

zoom(fig, option) Any of the above options can be specified on a figure other than the current figure using this syntax.

Remarks
zoom changes the axes limits by a factor of two (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.

See Also
linkaxes, pan

“Object Manipulation” for related functions
Index

Numerics
1-norm 2-1810

A
Accelerator
    Uimenu property 2-2370
activelegend 2-1692
allocation of storage (automatic) 2-2545
AlphaData
    surface property 2-2156
    surfaceplot property 2-2171
AlphaDataMapping
    patch property 2-1615
    surface property 2-2156
    surfaceplot property 2-2171
AmbientStrength
    Patch property 2-1615
    Surface property 2-2157
    surfaceplot property 2-2172
annotating plots 2-1693
archiving files 2-2547
arguments, M-file
    passing variable numbers of 2-2455
array
    product of elements 2-1741
    of random numbers 2-1799, 2-1801
    removing first n singleton dimensions of 2-1953
    removing singleton dimensions of 2-2022
    reshaping 2-1858
    shifting dimensions of 2-1953
    size of 2-1964
    sorting elements of 2-1974
    structure 2-1875, 2-1946
    sum of elements 2-2138
    swapping dimensions of 2-1672
    of all zeros 2-2545
arrays
    editing 2-2528
ASCII data
    converting sparse matrix after loading from 2-1984
    saving to disk 2-1899
aspect ratio of axes 2-1642
AutoScale
    quivergroup property 2-1783
AutoScaleFactor
    quivergroup property 2-1783
axes
    setting and querying limits 2-2531
    setting and querying plot box aspect ratio 2-1642
axes
    editing 2-1693
azimuth (spherical coordinates) 2-1992
azimuth of viewpoint 2-2468

B
BackFaceLighting
    Surface property 2-2157
    surfaceplot property 2-2172
BackFaceLightingpatch property 2-1616
BackgroundColor
    Uicontrol property 2-2339
BackgroundColor
    Text property 2-2230
badly conditioned 2-1810
Baseline
    stem property 2-2053
BaseValue
    stem property 2-2053
Index

BeingDeleted
  group property 2-2231
  quivergroup property 2-1783
  rectangle property 2-1824
  scatter property 2-1913
  stairseries property 2-2029
  stem property 2-2053
  surface property 2-2157
  surfaceplot property 2-2172
  transform property 2-1616
  Uipushtool property 2-2402
  Uitoggletool property 2-2424
  Uitoolbar property 2-2435

binary data
  saving to disk 2-1899

bold font
  TeX characters 2-2246

bubble plot (scatter function) 2-1909

Buckminster Fuller 2-2209

BusyAction
  patch property 2-1616
  quivergroup property 2-1783
  rectangle property 2-1824
  Root property 2-1878
  scatter property 2-1913
  stairseries property 2-2029
  stem property 2-2053
  Surface property 2-2158
  surfaceplot property 2-2173
  Text property 2-2232
  Uicontextmenu property 2-2324
  Uicontrol property 2-2340

C
  caching
    MATLAB directory 2-1634
  CallBack
    Uicontextmenu property 2-2324
    Uicontrol property 2-2340
    Uimenu property 2-2371
  CallbackObject, Root property 2-1878
  CaptureMatrix, Root property 2-1878
  CaptureRect, Root property 2-1878
  Cartesian coordinates 2-1702, 2-1992
  case
    in switch statement (defined) 2-2198
    lower to upper 2-2450
  Cayley-Hamilton theorem 2-1718

CData
  scatter property 2-1914
  Surface property 2-2158
  surfaceplot property 2-2173
  Uicontrol property 2-2341
  Uipushtool property 2-2403
  Uitoggletool property 2-2425

CDataMapping

Index-2
<table>
<thead>
<tr>
<th>Conversion</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cylindrical to Cartesian</td>
<td>2-1702</td>
</tr>
<tr>
<td>Full to sparse</td>
<td>2-1981</td>
</tr>
<tr>
<td>Lowercase to uppercase</td>
<td>2-2450</td>
</tr>
<tr>
<td>Partial fraction expansion to pole-residue</td>
<td>2-1860</td>
</tr>
<tr>
<td>Polar to Cartesian</td>
<td>2-1702</td>
</tr>
<tr>
<td>Pole-residue to partial fraction expansion</td>
<td>2-1860</td>
</tr>
<tr>
<td>Real to complex Schur form</td>
<td>2-1897</td>
</tr>
<tr>
<td>Spherical to Cartesian</td>
<td>2-1992</td>
</tr>
<tr>
<td>String to numeric array</td>
<td>2-2070</td>
</tr>
<tr>
<td>Conversion characters in format specification string</td>
<td>2-2011</td>
</tr>
<tr>
<td>Coordinate system and viewpoint</td>
<td>2-2468</td>
</tr>
<tr>
<td>Coordinates</td>
<td></td>
</tr>
<tr>
<td>Cartesian</td>
<td>2-1702, 2-1992</td>
</tr>
<tr>
<td>Cylindrical</td>
<td>2-1702</td>
</tr>
<tr>
<td>Polar</td>
<td>2-1702</td>
</tr>
<tr>
<td>Spherical</td>
<td>2-1992</td>
</tr>
<tr>
<td>CreateFcn</td>
<td></td>
</tr>
<tr>
<td>Patch property</td>
<td>2-1620</td>
</tr>
<tr>
<td>Quivergroup property</td>
<td>2-1785</td>
</tr>
<tr>
<td>Rectangle property</td>
<td>2-1825</td>
</tr>
<tr>
<td>Root property</td>
<td>2-1879</td>
</tr>
<tr>
<td>Scatter property</td>
<td>2-1915</td>
</tr>
<tr>
<td>Stairseries property</td>
<td>2-2030</td>
</tr>
<tr>
<td>Stemsseries property</td>
<td>2-2055</td>
</tr>
<tr>
<td>Surface property</td>
<td>2-2159</td>
</tr>
<tr>
<td>Surfaceplot property</td>
<td>2-2175</td>
</tr>
<tr>
<td>Text property</td>
<td>2-2232</td>
</tr>
<tr>
<td>Uicontextmenu property</td>
<td>2-2324</td>
</tr>
<tr>
<td>Uicontrol property</td>
<td>2-2341</td>
</tr>
<tr>
<td>Uimenu property</td>
<td>2-2372</td>
</tr>
<tr>
<td>Uipushtool property</td>
<td>2-2403</td>
</tr>
<tr>
<td>Uitoggletool property</td>
<td>2-2425</td>
</tr>
<tr>
<td>Uitoolbar property</td>
<td>2-2436</td>
</tr>
<tr>
<td>Cubic interpolation</td>
<td>2-1651</td>
</tr>
<tr>
<td>Current directory</td>
<td>2-1754</td>
</tr>
<tr>
<td>CurrentFigure, Root property</td>
<td>2-1879</td>
</tr>
<tr>
<td>Curvature, rectangle property</td>
<td>2-1826</td>
</tr>
<tr>
<td>Curve fitting (polynomial)</td>
<td>2-1711</td>
</tr>
<tr>
<td>Cuthill-McKee ordering, reverse</td>
<td>2-2207, 2-2209</td>
</tr>
<tr>
<td>Cylindrical coordinates</td>
<td>2-1702</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII, saving to disk</td>
<td>2-1899</td>
</tr>
<tr>
<td>Binary, saving to disk</td>
<td>2-1899</td>
</tr>
<tr>
<td>Computing 2-D stream lines</td>
<td>2-2076</td>
</tr>
<tr>
<td>Computing 3-D stream lines</td>
<td>2-2078</td>
</tr>
<tr>
<td>Formatting</td>
<td>2-2010</td>
</tr>
<tr>
<td>Reading from files</td>
<td>2-2249</td>
</tr>
<tr>
<td>Reducing number of elements in</td>
<td>2-1838</td>
</tr>
<tr>
<td>Smoothing 3-D</td>
<td>2-1972</td>
</tr>
<tr>
<td>Writing to strings</td>
<td>2-2010</td>
</tr>
<tr>
<td>Data, ASCII</td>
<td></td>
</tr>
<tr>
<td>Converting sparse matrix after loading from</td>
<td>2-1984</td>
</tr>
<tr>
<td>Debugging</td>
<td></td>
</tr>
<tr>
<td>M-files</td>
<td>2-1742</td>
</tr>
<tr>
<td>Decomposition</td>
<td></td>
</tr>
<tr>
<td>&quot;Economy-size&quot;</td>
<td>2-1759, 2-2193</td>
</tr>
<tr>
<td>Orthogonal-triangular (QR)</td>
<td>2-1759</td>
</tr>
<tr>
<td>Schur</td>
<td>2-1925</td>
</tr>
<tr>
<td>Singular value</td>
<td>2-1804, 2-2193</td>
</tr>
<tr>
<td>Definite integral</td>
<td>2-1770</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td></td>
</tr>
<tr>
<td>Quivergroup property</td>
<td>2-1785</td>
</tr>
<tr>
<td>Root property</td>
<td>2-1879</td>
</tr>
<tr>
<td>Scatter property</td>
<td>2-1915</td>
</tr>
<tr>
<td>Stairseries property</td>
<td>2-2031</td>
</tr>
<tr>
<td>Stem property</td>
<td>2-2055</td>
</tr>
</tbody>
</table>
Index

Surface property 2-2160
surfaceplot property 2-2175
Text property 2-2233, 2-2234
Uicontextmenu property 2-2325, 2-2342
Uimenu property 2-2373
Uipushtool property 2-2404
Uitoggletool property 2-2426
Uitoolbar property 2-2437
DeleteFcn, rectangle property 2-1826
DeleteFcnpatch property 2-1620
dependence, linear 2-2135
dependent functions 2-1742
derivative
polynomial 2-1708
detecting
positive, negative, and zero array elements 2-1958
diagonal
k-th (illustration) 2-2290
sparse 2-1986
dialog box
print 2-1739
question 2-1776
warning 2-2486
Diary, Root property 2-1879
DiaryFile, Root property 2-1879
differences
between sets 2-1944, 2-1945
differential equation solvers
ODE boundary value problems
extracting properties of 2-2288, 2-2289
parabolic-elliptic PDE problems 2-1657
DiffuseStrength
Surface property 2-2160
surfaceplot property 2-2176
DiffuseStrengthpatch property 2-1621
digamma function 2-1750
dimension statement (lack of in MATLAB) 2-2545
dimensions
size of 2-1964
direct term of a partial fraction expansion 2-1860
directories
listing MATLAB files in 2-2506
MATLAB
  caching 2-1634
  removing 2-1871
  removing from search path 2-1876
directory
temporary system 2-2216
directory, current 2-1754
discontinuities, eliminating (in arrays of phase angles) 2-2447
DisplayName
  quivergroup property 2-1785
  scatter property 2-1916
  stairseries property 2-2031
  stem property 2-2056
division
  remainder after 2-1854

E
Echo, Root property 2-1879
EdgeAlpha
  patch property 2-1621
  surface property 2-2160
  surfaceplot property 2-2176
EdgeColor
  patch property 2-1621
  Surface property 2-2161
  surfaceplot property 2-2176
  Text property 2-2233
EdgeColor, rectangle property 2-1826
Edgelighting
patch property 2-1622
Surface property 2-2161
surfaceplot property 2-2177
editable text 2-2330
eigenvalue
temporary approach to computation of 2-1706
problem 2-1709
problem, generalized 2-1709
problem, polynomial 2-1709
Wilkinson test matrix and 2-2519
eigenvector
matrix, generalized 2-1798
elevation (spherical coordinates) 2-1992
elevation of viewpoint 2-2468
Enable
Uicontrol property 2-2342
Uimenu property 2-2374
Uipushtool property 2-2404
Uitogglehtool property 2-2426
EraseMode
quivergroup property 2-1786
rectangle property 2-1826
scatter property 2-1916
stairseries property 2-2031
stem property 2-2056
Surface property 2-2162
surfaceplot property 2-2177
Text property 2-2234
EraseMode
patch property 2-1622
error messages
Out of memory 2-1596
ErrorMessage, Root property 2-1880
ErrorType, Root property 2-1880
escape characters in format specification string 2-2012
examples
reducing number of patch faces 2-1835
reducing volume data 2-1838
subsampling volume data 2-2141
Excel spreadsheets
loading 2-2534
executing statements repeatedly 2-2512
execution
improving speed of by setting aside storage 2-2545
pausing M-file 2-1641
time for M-files 2-1742
extension, filename
.mat 2-1899
Extent
Text property 2-2235
Uicontrol property 2-2343
F
FaceAlpha
patch property 2-1623
FaceAlphasurface property 2-2163
FaceAlphasurfaceplot property 2-2178
FaceColor
Surface property 2-2163
surfaceplot property 2-2179
FaceColor, rectangle property 2-1827
FaceColorpatch property 2-1624
FaceLighting
Surface property 2-2163
surfaceplot property 2-2179
FaceLightingpatch property 2-1624
faces, reducing number in patches 2-1834
Faces, patch property 2-1624
FaceVertexAlphaData, patch property 2-1625
FaceVertexCData, patch property 2-1626
factorization
QZ 2-1710, 2-1798
See also decomposition
factorization, Cholesky
	minimum degree ordering and (sparse) 2-2207
Figure
	redrawing 2-1840
figures
	annotating 2-1693
	saving 2-1904
filename

temporary 2-2217
filename extension
	.mat 2-1899
files
	compressing 2-2547
	Excel spreadsheets
	loading 2-2534
	fig 2-1904
figure, saving 2-1904
listing
	in directory 2-2506
locating 2-2509
mdl 2-1904
model, saving 2-1904
opening
	in Web browser 2-2500
opening in Windows applications 2-2520
pathname for 2-2509
reading

data from 2-2249
sound
	reading 2-2497
	writing 2-2499
.wav
	reading 2-2497
	writing 2-2499
WK1
	loading 2-2524
	writing to 2-2526
finding
	sign of array elements 2-1958
finish.m 2-1777
fixed-width font

text 2-2236
	uicontrols 2-2344
FixedWidthFontName, Root property 2-1880
floating-point arithmetic, IEEE
	smallest positive number 2-1814
flow control
	return 2-1865
	switch 2-2198
	while 2-2512
font

text 2-2236
	uicontrols 2-2344
FontSize

text 2-2236
	uicontrols 2-2344
FontUnits

text 2-2236
Uicontrol property 2-2344

FontWeight
Text property 2-2236
Uicontrol property 2-2345

ForegroundColor
Uicontrol property 2-2345
Uimenu property 2-2374

Format 2-1880
format
specification string, matching file data to 2-2024
FormatSpacing, Root property 2-1881
formatting data 2-2010
fraction, continued 2-1805
fragmented memory 2-1596
functions
locating 2-2509
pathname for 2-2509
that work down the first non-singleton dimension 2-1953

G
gamma function
logarithmic derivative 2-1750
Gaussian elimination
Gauss Jordan elimination with partial pivoting 2-1895
generalized eigenvalue problem 2-1709
geodesic dome 2-2209

graphics objects
Patch 2-1603
resetting properties 2-1857
Root 2-1877
setting properties 2-1938
Surface 2-2152
Text 2-2222

uicontextmenu 2-2320
Uicontrol 2-2329
Uimenu 2-2366

graphs
editing 2-1693
Greek letters and mathematical symbols 2-2244
GUIs, printing 2-1734

H
Hadamard matrix
subspaces of 2-2135

HandleVisibility
patch property 2-1627
quivergroup property 2-1787
rectangle property 2-1827
Root property 2-1881
scatter property 2-1917
stairseries property 2-2032
stem property 2-2057
Surface property 2-2164
surfaceplot property 2-2179
Text property 2-2236
uicontextmenu property 2-2326
Uicontrol property 2-2345
Uimenu property 2-2374
Uipushtool property 2-2405
Uitoggletool property 2-2427
Uitoolbar property 2-2437

HitTest
Patch property 2-1628
quivergroup property 2-1788
rectangle property 2-1828
Root property 2-1881
scatter property 2-1918
stairseries property 2-2033
stem property 2-2058
Surface property 2-2165
surfaceplot property 2-2180
Text property 2-2237
Uicontextmenu property 2-2326
Uicontrol property 2-2346
HitTestArea
  quivergroup property 2-1788
  scatter property 2-1918
  stairseries property 2-2034
  stem property 2-2058
HorizontalAlignment
  Text property 2-2238
  Uicontrol property 2-2346
HTML files
  opening 2-2500
hyperbolic
  secant 2-1931
  sine 2-1963
  tangent 2-2215
hyperplanes, angle between 2-2135

I
identity matrix
  sparse 2-1989
IEEE floating-point arithmetic
  smallest positive number 2-1814
indices, array
  of sorted elements 2-1975
integration
  polynomial 2-1714
  quadrature 2-1770
interpolated shading and printing 2-1735
Interpreter
  Text property 2-2239
Interruptible
  patch property 2-1628
quivergroup property 2-1788
rectangle property 2-1828
Root property 2-1881
scatter property 2-1918
stairseries property 2-2034
stem property 2-2058, 2-2061
Surface property 2-2165, 2-2181
Text property 2-2240
Uicontextmenu property 2-2326
Uicontrol property 2-2346
Uimenu property 2-2374
Uipushtool property 2-2406
Uitoggletool property 2-2428
Uitoolbar property 2-2437
involutary matrix 2-1602
italics font
  TeX characters 2-2246

J
Jacobi rotations 2-2008
Java version used by MATLAB 2-2464
jvm
  version used by MATLAB 2-2464

K
keyboard mode
  terminating 2-1865
KeyPressFcn
  Uicontrol property 2-2347

L
Label, Uimenu property 2-2375
labeling
  axes 2-2529
LaTeX, see TeX 2-2244
least squares
  polynomial curve fitting 2-1711
  problem, overdetermined 2-1679
limits of axes, setting and querying 2-2531
Line
  properties 2-1824
line
  editing 2-1693
linear dependence (of data) 2-2135
linear equation systems
  solving overdetermined 2-1761–2-1762
linear regression 2-1711
lines
  computing 2-D stream 2-2076
  computing 3-D stream 2-2078
  drawing stream lines 2-2080
LineStyle
  patch property 2-1629
  quivergroup property 2-1789
  rectangle property 2-1829
  stairseries property 2-2034
  stem property 2-2059
  surface object 2-2165
  surfaceplot object 2-2181
  text object 2-2240
LineWidth
  Patch property 2-1629
  quivergroup property 2-1789
  rectangle property 2-1829
  scatter property 2-1919
  stairseries property 2-2035
  stem property 2-2059
  Surface property 2-2166
  surfaceplot property 2-2181
  text object 2-2241
list boxes 2-2330

defining items 2-2351
ListBoxTop, Uicontrol property 2-2348
logarithm
  of real numbers 2-1812
logarithmic derivative
  gamma function 2-1750
logical operations
  XOR 2-2543
Lotus WK1 files
  loading 2-2524
  writing 2-2526
lower triangular matrix 2-2290
lowercase to uppercase 2-2450

M
machine epsilon 2-2513
Marker
  Patch property 2-1629
  quivergroup property 2-1789
  scatter property 2-1919
  stairseries property 2-2035
  stem property 2-2059
  Surface property 2-2166
  surfaceplot property 2-2181
MarkerEdgeColor
  Patch property 2-1629
  quivergroup property 2-1789
  scatter property 2-1919
  stairseries property 2-2036
  stem property 2-2060
  Surface property 2-2167
  surfaceplot property 2-2182
MarkerFaceColor
  Patch property 2-1630
  quivergroup property 2-1790
  scatter property 2-1920
  stairseries property 2-2035
  stem property 2-2059
  Surface property 2-2166
  surfaceplot property 2-2181

stairseries property 2-2036
stem property 2-2060
Surface property 2-2167
surfaceplot property 2-2183
MarkerSize
Patch property 2-1630
quivergroup property 2-1790
stairseries property 2-2036
stem property 2-2061
Surface property 2-2167
surfaceplot property 2-2183
MAT-file 2-1899
converting sparse matrix after loading from 2-1984
MAT-files
listing for directory 2-2506
MATLAB
quitting 2-1777
version number, displaying 2-2458
MATLAB startup file 2-2043
matlab.mat 2-1899
matrices
preallocation 2-2545
matrix
complex unitary 2-1759
condition number of 2-1810
converting to from string 2-2023
decomposition 2-1759
Hadamard 2-2135
Hermitian Toeplitz 2-2284
involuntary 2-1602
lower triangular 2-2290
magic squares 2-2138
orthonormal 2-1759
Pascal 2-1602, 2-1717
permutation 2-1759
pseudoinverse 2-1679
reduced row echelon form of 2-1895
replicating 2-1856
rotating 90° 2-1890
Schur form of 2-1897, 2-1925
sorting rows of 2-1977
sparse See sparse matrix
square root of 2-2019
subspaces of 2-2135
Toeplitz 2-2284
trace of 2-2285
unitary 2-2193
upper triangular 2-2296
Vandermonde 2-1713
Wilkinson 2-1987, 2-2519
writing to spreadsheet 2-2526
Max, Uicontrol property 2-2348
MaxHeadSize property 2-2348
memory
minimizing use of 2-1596
variables in 2-2516
mesh plot
tetrahedron 2-2218
MeshStyle, Surface property 2-2167
MeshStyle, surfaceplot property 2-2183
message
error See error message
warning See warning message
methods
locating 2-2509
MEX-files
listing for directory 2-2506
M-file
pausing execution of 2-1641
M-files
creating
in MATLAB directory 2-1634
debugging with profile 2-1742
listing names of in a directory 2-2506
optimizing 2-1742
Microsoft Excel files
loading 2-2534
Min, UIControl property 2-2349
minimum degree ordering 2-2207
models
saving 2-1904
MonitorPosition
Root property 2-1881
Moore-Penrose pseudoinverse 2-1679
multidimensional arrays
rearranging dimensions of 2-1672
removing singleton dimensions of 2-2022
reshaping 2-1858
size of 2-1964
sorting elements of 2-1974
random 2-1799, 2-1801
real 2-1811
smallest positive 2-1814
numerical evaluation
triple integral 2-2292

O
OffCallback
   Uicontrol property 2-2429
OnCallback
   Uicontrol property 2-2429
opening
   files in Windows applications 2-2520
operating system command 2-2212
optimizing M-file execution 2-1742
ordering
   minimum degree 2-2207
   reverse Cuthill-McKee 2-2207, 2-2209
orthogonal-triangular decomposition 2-1759
orthonormal matrix 2-1759
Out of memory (error message) 2-1596
overdetermined equation systems, solving 2-1761–2-1762

N
NaN (Not-a-Number)
   returned by rem 2-1854
nonzero entries
   specifying maximum number of in sparse matrix 2-1981
nonzero entries (in sparse matrix)
   replacing with ones 2-2001
norm
   1-norm 2-1810
   pseudoinverse and 2-1679–2-1681
NormalMode
   Patch property 2-1631
   Surface property 2-2167
   surfaceplot property 2-2183
numbers
   prime 2-1724
pack 2-1596
pagesetupdlg 2-1598
Parent
   Patch property 2-1631
   quivergroup property 2-1791
   rectangle property 2-1829
   Root property 2-1882
   scatter property 2-1920
   stairseries property 2-2036
   stem property 2-2061
Surface property 2-2168
surfaceplot property 2-2183
Text property 2-2243
Uicontrol property 2-2349
Uimenu property 2-2376
Uipushtool property 2-2407
Uitoggletool property 2-2429
Uitoolbar property 2-2438
pareto 2-1600
partial fraction expansion 2-1860
partialpath 2-1601
pascal 2-1602
Pascal matrix 2-1602, 2-1717
Patch
  converting a surface to 2-2150
  creating 2-1603
  defining default properties 2-1609
  properties 2-1615
  reducing number of faces 2-1834
  reducing size of face 2-1955
patch 2-1603
path
  current 2-1634
  removing directories from 2-1876
  viewing 2-1639
path 2-1634
path2rc 2-1636
pathdef 2-1637
pathname
  partial 2-1601
pathnames
  of functions or files 2-2509
  relative 2-1601
pathsep 2-1638
pathtool 2-1639
pause 2-1641
pausing M-file execution 2-1641
pbaspect 2-1642
pcg 2-1647
pcg 2-1647
pchlp 2-1651
pcode 2-1653
pcolor 2-1654
PDE See Partial Differential Equations
pdepe 2-1657
pdeval 2-1668
perl 2-1670
Perl scripts in MATLAB 2-1670
perms 2-1671
permutation
  of array dimensions 2-1672
  matrix 2-1759
  random 2-1803
permutations of n elements 2-1671
permute 2-1672
persistent 2-1673
persistent variable 2-1673
phase, complex
  correcting angles 2-2446
pi 2-1674
pie 2-1675
pie3 2-1677
pinv 2-1679
planerot 2-1682
playshow function 2-1683
plot 2-1684
  editing 2-1693
plot box aspect ratio of axes 2-1642
plot editing mode
  overview 2-1694
Plot Editor
  interface 2-1694, 2-1748
plot, volumetric
slice plot 2-1967
plot3 2-1690
plotedit 2-1692
plotmatrix 2-1696
plotting
  2-D plot 2-1684
  3-D plot 2-1690
plot with two y-axes 2-1699
ribbon plot 2-1868
rose plot 2-1887
scatter plot 2-1696
scatter plot, 3-D 2-1911
semilogarithmic plot 2-1934
stem plot, 3-D 2-2051
surface plot 2-2145
volumetric slice plot 2-1967
plotting See visualizing
plotyy 2-1699
PointerLocation, Root property 2-1882
PointerWindow, Root property 2-1882
pol2cart 2-1702
polar 2-1703
polar coordinates 2-1702
  converting to cylindrical or Cartesian 2-1702
poles of transfer function 2-1860
poly 2-1705
polyarea 2-1707
polyder 2-1708
polyeig 2-1709
polyfit 2-1711
polygamma function 2-1750
polygon
  area of 2-1707
  creating with patch 2-1603
polyint 2-1714
polynomial
  analytic integration 2-1714
characteristic 2-1705–2-1706, 2-1885
coefficients (transfer function) 2-1860
curve fitting with 2-1711
derivative of 2-1708
eigenvalue problem 2-1709
evaluation 2-1715
evaluation (matrix sense) 2-1717
polyval 2-1715
polyvalm 2-1717
pop-up menus 2-2330
  defining choices 2-2351
Position
  Text property 2-2243
  Uicontextmenu property 2-2327
  Uicontrol property 2-2350
  Uimenu property 2-2376
Position, rectangle property 2-1829
PostScript
  printing interpolated shading 2-1735
pow2 2-1719
power
  of real numbers 2-1815
ppval 2-1720
preallocation
  matrix 2-2545
prefdir 2-1721
preferences
  opening the dialog box 2-1723
prime numbers 2-1724
primes 2-1724
print 2-1721
printdlg 2-1739
printer drivers
  GhostScript drivers 2-1726
  interpolated shading 2-1735
  MATLAB printer drivers 2-1726
printing
GUIs 2-1734
interpolated shading 2-1735
on MS-Windows 2-1733
with a variable filename 2-1736
with non-normal EraseMode 2-1623, 2-1827, 2-2162, 2-2178, 2-2235
printing tips 2-1733
printopt 2-1725
printpreview 2-1740
prod 2-1741
product
  of array elements 2-1741
profile 2-1742
profsave 2-1747
proppedit 2-1742
proppanel 2-1749
pseudoinverse 2-1679
psi 2-1750
publish function 2-1752
push buttons 2-2331
pwd 2-1754
quit 2-1777
quitting MATLAB 2-1777
quiver 2-1778
quiver3 2-1781
qz 2-1798
QZ factorization 2-1710, 2-1798
radio buttons 2-2331
rand 2-1799
randn 2-1801
random
  numbers 2-1799, 2-1801
  permutation 2-1803
  sparse matrix 2-2006, 2-2007
  symmetric sparse matrix 2-2008
randperm 2-1803
rank 2-1804
rank of a matrix 2-1804
rat 2-1805
rational fraction approximation 2-1805
rats 2-1805
rbbox 2-1808, 2-1840
rcond 2-1810
reading
  data from files 2-2249
  formatted data from strings 2-2023
readme files, displaying 2-2508
real 2-1811
real numbers 2-1811
reallog 2-1812
realmax 2-1813
realmin 2-1814
realpow 2-1815
realsqrt 2-1816
rearranging arrays

Q
qmr 2-1755
qr 2-1759
QR decomposition 2-1759
  deleting column from 2-1763
qrdelete 2-1763
qrinsert 2-1765
qrupdate 2-1767
quad 2-1770
quad8 2-1770
quad1 2-1772
quadrature 2-1770
quadv 2-1774
questdlg 2-1776
Index

removing first \( n \) singleton dimensions 2-1953
removing singleton dimensions 2-2022
reshaping 2-1858
shifting dimensions 2-1953
swapping dimensions 2-1672
rearranging matrices
  rotating \( 90^\circ \) 2-1890
rectint 2-1831
RecursionLimit
  Root property 2-1882
recycle 2-1832
reduced row echelon form 2-1895
reducespatch 2-1834
reducevolume 2-1838
refresh 2-1840
regexp 2-1843
regexpi 2-1843
regexprep 2-1849
regression
  linear 2-1711
rehash 2-1852
rem 2-1854
remainder after division 2-1854
repeatedly executing statements 2-2512
replicating a matrix 2-1856
repmat 2-1856
reshape 2-1858
residue 2-1860
residues of transfer function 2-1860
restoredefaultpath 2-1863
rethrow 2-1864
return 2-1865
reverse Cuthill-McKee ordering 2-2207, 2-2209
RGB, converting to HSV 2-1866
rgb2gray 2-1866
rgbplot 2-1867
ribbon 2-1868
right-click and context menus 2-2320
rmdir 2-1871
rmdir (ftp) 2-1874
rmfield 2-1875
rmpath 2-1876
root 2-1877
Root graphics object 2-1877
root object 2-1877
root, see rootobject 2-1877
roots 2-1885
roots of a polynomial 2-1705–2-1706, 2-1885
rose 2-1887
rosser 2-1889
rot90 2-1890
rotate 2-1891
rotate3d 2-1893
Rotation, Text property 2-2243
rotations
  Jacobi 2-2008
round
  to nearest integer 2-1894
round 2-1894
roundoff error
  characteristic polynomial and 2-1706
  partial fraction expansion and 2-1861
  polynomial roots and 2-1885
  sparse matrix conversion and 2-1985
rref 2-1895
rrefmovie 2-1895
rsf2csf 2-1897
rubberband box 2-1808

S
save 2-1899
saveas 2-1904
saveobj 2-1907
savepath 2-1908
saving
  ASCII data 2-1899
  workspace variables 2-1899
schur 2-1925
Schur decomposition 2-1925
Schur form of matrix 2-1897, 2-1925
ScreenDepth, Root property 2-1882
ScreenSize, Root property 2-1883
script 2-1927
search path 2-1876
  MATLAB's 2-1634
  modifying 2-1639
  viewing 2-1639
sec 2-1928
secant 2-1928
  hyperbolic 2-1931
secd 2-1930
sech 2-1931
Selected
  Patch property 2-1631
  quivergroup property 2-1791
  rectangle property 2-1830
  Root property 2-1883
  scatter property 2-1920
  stairseries property 2-2036
  stem property 2-2061
  Surface property 2-2168
  surfaceplot property 2-2184
  Text property 2-2244
  Uicontextmenu property 2-2328
  Uicontrol property 2-2350
selectmoveresize 2-1935
semilogx 2-1934
semilogy 2-1934
sendmail 2-1936
Separator
  Uipushtool property 2-2407
  Uitoggletool property 2-2429
Separator, Uimenu property 2-2376
set 2-1938
set
  timer object 2-1941
set operations
  difference 2-1944, 2-1945
  exclusive or 2-1949
  union 2-2441
  unique 2-2442
setdiff 2-1945
setfield 2-1946
setstr 2-1948
setxor 2-1949
shading 2-1944
shading colors in surface plots 2-1950
shell script 2-2212, 2-2444
shiftdim 2-1953
ShowArrowHead
  quivergroup property 2-1791
ShowHiddenHandles, Root property 2-1883
shrinkfaces 2-1955
shutdown 2-1777
sign 2-1958
signum function 2-1958
Simpson’s rule, adaptive recursive 2-1771
Simulink
  version number, displaying 2-2458
sin 2-1959
sind 2-1961
sine 2-1959
hyperbolic 2-1963
single 2-1962
singular value
  decomposition 2-1804, 2-2193
  rank and 2-1804
sinh 2-1963
size
  array dimensions 2-1964
size of array dimensions 2-1964
size of fonts, see also FontSize property 2-2246
size vector 2-1858
SizeData
  scatter property 2-1921
slice 2-1967
sliders 2-2331
SliderStep, Uicontrol property 2-2350
smooth3 2-1972
smoothing 3-D data 2-1972
soccer ball (example) 2-2209
sort 2-1974
sorting
  array elements 2-1974
  matrix rows 2-1977
sortrows 2-1977
sound
  converting vector into 2-1978, 2-1979
  files
    reading 2-2497
    writing 2-2499
  playing 2-2495
  recording 2-2498
  resampling 2-2495
  sampling 2-2498
sound 2-1978, 2-1979
source control systems
  undo checkout 2-2440
spalloc 2-1980
sparse 2-1981
sparse matrix
  allocating space for 2-1980
  applying function only to nonzero elements of
    2-1990
  diagonal 2-1986
  identity 2-1989
  random 2-2006, 2-2007
  random symmetric 2-2008
  replacing nonzero elements of with ones 2-2001
  results of mixed operations on 2-1982
  solving least squares linear system 2-1760
  specifying maximum number of nonzero
    elements 2-1981
  visualizing sparsity pattern of 2-2016
spaugment 2-1983
spconvert 2-1984
spdiags 2-1986
SpecularColorReflectance
  Patch property 2-1631
  Surface property 2-2168
  surfaceplot property 2-2184
SpecularExponent
  Patch property 2-1632
  Surface property 2-2168
  surfaceplot property 2-2184
SpecularStrength
  Patch property 2-1632
  Surface property 2-2168
surfaceplot property 2-2184
spye 2-1989
spfun 2-1990
sph2cart 2-1992
sphere 2-1993
spherical coordinates 2-1992
spinmap 2-1995
spline 2-1996
spones 2-2001
spparms 2-2002
sprand 2-2006
sprandn 2-2007
sprandsym 2-2008
sprank 2-2009
spreadsheets
  loading WK1 files 2-2524
  loading XLS files 2-2534
  writing from matrix 2-2526
sprintf 2-2010
sqrt 2-2018
sqrtm 2-2019
square root
  of a matrix 2-2019
  of array elements 2-2018
  of real numbers 2-1816
squeeze 2-2022
sscanf 2-2023
standard deviation 2-2044
start
  timer object 2-2040
startat
  timer object 2-2041
startup 2-2043
startup file 2-2043
State
  Uitoggletool property 2-2429
static text 2-2331
std 2-2044
stem3 2-2051
stop
  timer object 2-2065
stopwatch timer 2-2271
storage
  sparse 2-1981
  storage allocation 2-2545
str2double 2-2066
str2func 2-2067
str2mat 2-2069
str2num 2-2070
strcat 2-2071
strcmp 2-2073
strcmpi 2-2075
stream lines
  computing 2-D 2-2076
  computing 3-D 2-2078
  drawing 2-2080
stream2 2-2076
stream3 2-2078
strfind 2-2101
String
  Text property 2-2244
  Uicontrol property 2-2351
string
  comparing one to another 2-2073
  comparing the first $n$ characters of two 2-2107
  converting to numeric array 2-2070
  converting to uppercase 2-2450
  dictionary sort of 2-1977
  finding first token in 2-2117
  searching and replacing 2-2116
strings
  converting to matrix (formatted) 2-2023
  writing data to 2-2010
strings 2-2103
| strjust  | 2-2105 |
| strmatch | 2-2106 |
| strncmp  | 2-2107 |
| strncmpi | 2-2108 |
| strrread | 2-2109 |
| strrep   | 2-2116 |
| strtok   | 2-2117 |
| strtrim  | 2-2120 |
| struct   | 2-2121 |
| struct2cell | 2-2123 |
| structure array |  remove field from 2-1875 |
|          |  setting contents of a field of 2-1946 |
| strvcat  | 2-2124 |
| Style    |  |
|          | Uicontrol property 2-2352 |
| sub2ind  | 2-2125 |
| subplot  | 2-2127 |
| subsasgn | 2-2133 |
| subscripts |  in axis title 2-2281 |
|          |  in text strings 2-2246 |
| subsindex | 2-2134 |
| subspace | 2-2135 |
| subsref  | 2-2136 |
| substruct | 2-2137 |
| subvolume | 2-2141 |
| sum      | of array elements 2-2138 |
| sum      | 2-2138 |
| superiorto | 2-2143 |
| superscripts |  in axis title 2-2282 |
|          |  in text strings 2-2246 |
| support  | 2-2143 |
| surf     | 2-2145 |
| surf2patch | 2-2150 |
| Surface  |  converting to a patch 2-2150 |
|          |  creating 2-2152 |
|          |  defining default properties 2-1821, 2-2155 |
|          |  properties 2-2156, 2-2171 |
| surface  | 2-2152 |
| surfe    | 2-2145 |
| surf1    | 2-2188 |
| surfnorm | 2-2191 |
| svd      | 2-2193 |
| svds     | 2-2196 |
| switch   | 2-2198 |
| symamd   | 2-2200 |
| symbfact | 2-2202 |
| symbols  | in text 2-2244 |
| symmlq   | 2-2203 |
| symmdm   | 2-2207 |
| symrcm   | 2-2209 |
| system   | 2-2212 |
| system directory, temporary | 2-2216 |
| T        |  |
| Tag      |  Patch property 2-1632 |
|          |  quivergroup property 2-1791 |
|          |  rectangle property 2-1830 |
|          |  Root property 2-1883 |
|          |  scatter property 2-1921 |
|          |  stairseries property 2-2037 |
|          |  stem property 2-2061 |
|          |  Surface property 2-2169 |
|          |  surfaceplot property 2-2184 |
|          |  Text property 2-2247 |
|          |  Uicontextmenu property 2-2328 |
|          |  Uicontrol property 2-2352 |
|          |  Uimenu property 2-2376 |
Uipushtool property 2-2407
Uitoggletool property 2-2429
Uitoolbar property 2-2438

tan 2-2213
tand 2-2214
tangent 2-2213
    hyperbolic 2-2215
tanh 2-2215
tempdir 2-2216
tempname 2-2217
temporary
    files 2-2217
    system directory 2-2216
terminating MATLAB 2-1777
tetrahedron
    mesh plot 2-2218
tetramesh 2-2218
TeX commands in text 2-2244

Text
    creating 2-2222
    defining default properties 2-2225
    fixed-width font 2-2236
    properties 2-2230

text
    subscripts 2-2246
    superscripts 2-2246
text 2-2222
    editing 2-1693
textrad 2-2249
textscan 2-2255
textwrap 2-2270
tic 2-2271

tiling (copies of a matrix) 2-1856

time
    elapsed (stopwatch timer) 2-2271
timer
    properties 2-2272
timer
    timer object 2-2272
timerfind
    timer object 2-2277
timerfindall
    timer object 2-2279
title
    with superscript 2-2281, 2-2282
    title 2-2271
toc 2-2271
todatenum 2-2283
toeplitz 2-2284
    Toeplitz matrix 2-2284
    toggle buttons 2-2331
token See also string 2-2117

TooltipString
    Uicontrol property 2-2352
    Uipushtool property 2-2407
    Uitoggletool property 2-2429
trace 2-2285
    trace of a matrix 2-2285
trapz 2-2286


treelayout 2-2288
treeplot 2-2289
triangulation
    2-D plot 2-2294
tril 2-2290

trimesh 2-2283
triple integral
    numerical evaluation 2-2292

triplequad 2-2292
triplot 2-2294

trisurf 2-2295

triu 2-2296

true 2-2297

try 2-2298
tsearch 2-2299
tsearchn 2-2300
Type
  Patch property 2-1632
  quivergroup property 2-1792
  rectangle property 2-1830
  Root property 2-1883
  scatter property 2-1921
  stairseries property 2-2037
  stem property 2-2062
  Surface property 2-2169
  surfaceplot property 2-2185
  Text property 2-2247
  Uicontextmenu property 2-2328
  Uicontrol property 2-2352
  Uimenu property 2-2376
  Uipushtool property 2-2407
  Uitoggletool property 2-2430
  Uitoolbar property 2-2439

U
  UData
    quivergroup property 2-1792
  UDatasource
    quivergroup property 2-1793
  UIButtonGroup
    defining default properties 2-2308
  uibuttongroup function 2-2301
  UIButtonGroup Properties 2-2308
  UIContextMenu
    Patch property 2-1632
    quivergroup property 2-1792
    rectangle property 2-1830
    scatter property 2-1921
    stairseries property 2-2037
    stem property 2-2062
    Surface property 2-2169
  surfaceplot property 2-2185
  Text property 2-2247
  UIContextMenu
    Uicontrol property 2-2352
    Uicontextmenu
      Uicontextmenu property 2-2328
      uicontextmenu 2-2320
    Uicontextmenu Properties 2-2323
    Uicontrol
      defining default properties 2-2337
      fixed-width font 2-2344
      types of 2-2329
      uicontrol 2-2329
    Uicontrol Properties 2-2337
    uigetdir 2-2355
    uigetfile 2-2359
    uiimport 2-2365
    Uimenu
      creating 2-2366
      defining default properties 2-2369
      Properties 2-2369
    uimenu 2-2366
    Uimenu Properties 2-2369
    uint8 2-2377
    uint8, uint16, uint32, uint64 2-2377
    uiopen 2-2379
    Uipanel
      defining default properties 2-2385
      uipanel function 2-2380
      Uipanel Properties 2-2385
    Uipushtool
      defining default properties 2-2401
    uipushtool 2-2397
    Uipushtool Properties 2-2401
    uiputfile 2-2408
    uiresume 2-2413
    uisave 2-2414
uisetcolor 2-2415
uisetfont 2-2416
uistack 2-2418
Uitoggletool
  defining default properties 2-2423
uitoggletool 2-2419
Uitoggletool Properties 2-2423
Uitoolbar
  defining default properties 2-2434
uitoolbar 2-2431
Uitoolbar Properties 2-2434
uiwait 2-2413
uncompress files 2-2449
undocheckout 2-2440
union 2-2441
unique 2-2442
unitary matrix (complex) 2-1759
Units
  Root property 2-1883
  Text property 2-2247
  Uicontrol property 2-2353
unix 2-2444
unmkpp 2-2445
unwrap 2-2446
unzip 2-2449
upper 2-2450
upper triangular matrix 2-2296
url
  opening in Web browser 2-2500
urllread 2-2451
UserData
  Patch property 2-1633
  quivergroup property 2-1792
  rectangle property 2-1830
  Root property 2-1884
  scatter property 2-1922
  stairseries property 2-2038
stem property 2-2062
Surface property 2-2169
surfaceplot property 2-2185
Text property 2-2247
Uicontextmenu property 2-2328
Uicontrol property 2-2353
Uimenu property 2-2376
Uipushtool property 2-2407
Uitoggletool property 2-2430
Uitoolbar property 2-2439

V
Value, Uicontrol property 2-2353
vander 2-2453
Vandermonde matrix 2-1713
var 2-2454
vargout 2-2455
variable numbers of M-file arguments 2-2455
variables
  graphical representation of 2-2528
  in workspace 2-2528
  listing 2-2516
  persistent 2-1673
  saving 2-1899
  sizes of 2-2516
VData
  quivergroup property 2-1793
VDataSource
  quivergroup property 2-1793
vectorize 2-2457
vectorize 2-2457
ver 2-2458
verctrl 2-2460
version 2-2464
version numbers
  displaying 2-2458
Index-24

returned as strings 2-2464
vertcat 2-2465
VertexNormals
   Patch property 2-1633
   Surface property 2-2169
   surfaceplot property 2-2185
VerticalAlignment, Text property 2-2248
Vertices, Patch property 2-1633
view
   azimuth of viewpoint 2-2468
   coordinate system defining 2-2468
   elevation of viewpoint 2-2468
view 2-2467
viewmtx 2-2470
Visible
   Patch property 2-1633
   quivergroup property 2-1792
   rectangle property 2-1830
   Root property 2-1884
   scatter property 2-1922
   stairseries property 2-2038
   stem property 2-2062
   Surface property 2-2169
   surfaceplot property 2-2185
   Text property 2-2248
   Uicontextmenu property 2-2328
   Uicontrol property 2-2354
   Umenu property 2-2376
   Uipushtool property 2-2407
   Uitoggletool property 2-2430
   Uiuiolbar property 2-2439
visualizing
   sparse matrices 2-2016
volumes
   computing 2-D stream lines 2-2076
   computing 3-D stream lines 2-2078
   drawing stream lines 2-2080
reducing face size in isosurfaces 2-1955
reducing number of elements in 2-1838
voronoi 2-2476
Voronoi diagrams
   multidimensional visualization 2-2479
   two-dimensional visualization 2-2476
voronoin 2-2479

W
wait
   timer object 2-2481
waitbar 2-2482
waitfor 2-2482
waitforbuttonpress 2-2485
warnldlg 2-2486
warning 2-2487
   warning message (enabling, suppressing, and
   displaying) 2-2487
waterfall 2-2486
.wav files
   reading 2-2497
   writing 2-2499
waverecord 2-2498
wavfinfo 2-2494
wavplay 2-2495
wavplay 2-2495
wavread 2-2494, 2-2497
wavrecord 2-2498
wavwrite 2-2499
WData
   quivergroup property 2-1794
WDataSource
   quivergroup property 2-1794
web 2-2500
Web browser
   pointing to file or url 2-2500
weekday 2-2504
well conditioned 2-1810
what 2-2506
whatsnew 2-2508
which 2-2509
while 2-2512
white space characters, ASCII 2-2117
whitebg 2-2515
wilkinson 2-2519
Wilkinson matrix 2-1987, 2-2519
winopen 2-2520
winqueryreg 2-2521
WK1 files
  loading 2-2524
  writing from matrix 2-2526
wk1finfo 2-2523
wk1read 2-2524
wk1write 2-2526
workspace
  consolidating memory 2-1596
  predefining variables 2-2043
  saving 2-1899
  variables in 2-2516
  viewing contents of 2-2528
workspace 2-2528

X
x-axis limits, setting and querying 2-2531
XData
  Patch property 2-1633
  quivergroup property 2-1794
  scatter property 2-1922
  stairseries property 2-2038
  stem property 2-2062
  Surface property 2-2170
  surfaceplot property 2-2185
XDataMode
  quivergroup property 2-1795
  stairseries property 2-2038
  stem property 2-2063
  surfaceplot property 2-2185
XDataSource
  area property 2-1922
  quivergroup property 2-1795
  stairseries property 2-2038
  stem property 2-2063
  surfaceplot property 2-2186
xlabel 2-2529
xlim 2-2531
XLS files
  loading 2-2534
  xlsfinfo 2-2533
  xlsread 2-2534
  xlsxwrite 2-2538
  xmlread 2-2541
  xmlwrite 2-2542
  logical XOR 2-2543
xor 2-2543
XOR, printing 2-1623, 2-1786, 2-1827, 2-1917, 2-2032, 2-2056, 2-2162, 2-2178, 2-2235
xslt 2-2544
xyz coordinates See Cartesian coordinates

Y
y-axis limits, setting and querying 2-2531
YData
  Patch property 2-1633
  quivergroup property 2-1795
  scatter property 2-1923
  stairseries property 2-2039
  stem property 2-2063
  Surface property 2-2170

Index-25
surfaceplot property 2-2186
YDataMode
  quivergroup property 2-1796
  surfaceplot property 2-2186
YDataSource
  quivergroup property 2-1796
  scatter property 2-1923
  stairseries property 2-2039
  stem property 2-2063
  surfaceplot property 2-2186
ylabel 2-2529
ylim 2-2531

Z
z-axis limits, setting and querying 2-2531
ZData
  lineseries property 2-1923
  Patch property 2-1633
  quivergroup property 2-1796
  stemseries property 2-2064
  Surface property 2-2170
  surfaceplot property 2-2187
ZDataSource
  lineseries property 2-1923, 2-2064
  surfaceplot property 2-2187
zeros 2-2545
zip 2-2547
zlabel 2-2529
zlim 2-2531
zoom 2-2549