How to Contact The MathWorks:

www.mathworks.com        Web
comp.soft-sys.matlab     Newsgroup

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

508-647-7000            Phone
508-647-7001            Fax

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

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

MATLAB Function Reference Volume 1: A - E
© COPYRIGHT 1984 - 2004 by The MathWorks, Inc.
The software described in this document is furnished under a license agreement. The software may be used
or copied only under the terms of the license agreement. No part of this manual may be photocopied or repro-
duced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by,
for, or through the federal government of the United States. By accepting delivery of the Program or
Documentation, the government hereby agrees that this software or documentation qualifies as commercial
computer software or commercial computer software documentation as such terms are used or defined in
FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this
Agreement and only those rights specified in this Agreement, shall pertain to and govern the use,
modification, reproduction, release, performance, display, and disclosure of the Program and Documentation
by the federal government (or other entity acquiring for or through the federal government) and shall
supersede any conflicting contractual terms or conditions. If this License fails to meet the government's
needs or is inconsistent in any respect with federal procurement law, the government agrees to return the
Program and Documentation, unused, to The MathWorks, Inc.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and
TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:  December 1996  First printing  For MATLAB 5.0 (Release 8)
          June 1997    Online only  Revised for MATLAB 5.1 (Release 9)
          October 1997  Online only  Revised for MATLAB 5.2 (Release 10)
          January 1999  Online only  Revised for MATLAB 5.3 (Release 11)
          June 1999    Second printing  For MATLAB 5.3 (Release 11)
          June 2001    Online only  Revised for MATLAB 6.1 (Release 12.1)
          July 2002    Online only  Revised for 6.5 (Release 13)
          June 2004    Online only  Revised for 7.0 (Release 14)
## Functions — Categorical List

### Desktop Tools and Development Environment .......................... 1-2
- Startup and Shutdown .............................................. 1-2
- Command Window and History .................................... 1-3
- Help for Using MATLAB .......................................... 1-3
- Workspace, Search Path, and File Operations .................... 1-3
  - Workspace ......................................................... 1-4
  - Search Path ....................................................... 1-4
  - File Operations .................................................. 1-4
- Programming Tools ................................................ 1-5
  - Editing and Debugging ......................................... 1-5
  - Performance Improvement and Tuning Tools and Techniques .. 1-5
- Source Control ..................................................... 1-6
- Publishing .......................................................... 1-6
- System .............................................................. 1-6

### Mathematics .......................................................... 1-7
- Arrays and Matrices ................................................ 1-8
  - Basic Information ............................................... 1-8
  - Operators ......................................................... 1-8
  - Operations and Manipulation ................................. 1-9
  - Elementary Matrices and Arrays ............................ 1-10
  - Specialized Matrices .......................................... 1-10
- Linear Algebra ...................................................... 1-10
  - Matrix Analysis ................................................ 1-11
  - Linear Equations ............................................... 1-11
  - Eigenvalues and Singular Values ............................ 1-11
  - Matrix Logarithms and Exponentials ......................... 1-12
  - Factorization .................................................... 1-12
- Elementary Math .................................................... 1-12
  - Trigonometric ................................................... 1-13
  - Exponential ....................................................... 1-14
  - Complex .......................................................... 1-14
  - Rounding and Remainder ....................................... 1-14
Discrete Math (e.g., Prime Factors) ......................... 1-15
Data Analysis and Fourier Transforms .................... 1-15
  Basic Operations ............................................. 1-15
  Finite Differences ......................................... 1-15
  Correlation ..................................................... 1-16
  Filtering and Convolution ................................ 1-16
  Fourier Transforms ........................................ 1-16
Polynomials ..................................................... 1-16
Interpolation and Computational Geometry ................ 1-17
  Interpolation ................................................... 1-17
  Delaunay Triangulation and Tessellation ............... 1-17
  Convex Hull .................................................. 1-18
  Voronoi Diagrams .......................................... 1-18
  Domain Generation ......................................... 1-18
Coordinate System Conversion ................................ 1-18
  Cartesian ..................................................... 1-18
Nonlinear Numerical Methods ................................ 1-18
  Ordinary Differential Equations (IVP) ................. 1-19
  Delay Differential Equations ......................... 1-19
  Boundary Value Problems ................................. 1-19
  Partial Differential Equations ......................... 1-19
  Optimization ............................................... 1-19
  Numerical Integration (Quadrature) .................... 1-20
Specialized Math ............................................... 1-20
  Sparse Matrices ............................................ 1-20
    Elementary Sparse Matrices ......................... 1-21
    Full to Sparse Conversion ......................... 1-21
    Working with Sparse Matrices .................... 1-21
    Reordering Algorithms .............................. 1-21
    Linear Algebra .......................................... 1-21
    Linear Equations (Iterative Methods) ............. 1-22
    Tree Operations ........................................ 1-22
Math Constants .............................................. 1-22

Programming and Data Types ............................... 1-23
  Data Types .................................................. 1-23
    Numeric ..................................................... 1-24
    Characters and Strings ................................ 1-24
    Structures ................................................ 1-26
    Cell Arrays ............................................... 1-26
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio and Audio/Video</td>
<td>1-41</td>
</tr>
<tr>
<td>General</td>
<td>1-41</td>
</tr>
<tr>
<td>SPARCstation-Specific Sound Functions</td>
<td>1-41</td>
</tr>
<tr>
<td>Microsoft WAVE Sound Functions</td>
<td>1-41</td>
</tr>
<tr>
<td>Audio/Video Interleaved (AVI) Functions</td>
<td>1-41</td>
</tr>
<tr>
<td>Images</td>
<td>1-41</td>
</tr>
<tr>
<td>Internet Exchange</td>
<td>1-42</td>
</tr>
<tr>
<td><strong>Graphics</strong></td>
<td>1-43</td>
</tr>
<tr>
<td>Basic Plots and Graphs</td>
<td>1-43</td>
</tr>
<tr>
<td>Plotting Tools</td>
<td>1-43</td>
</tr>
<tr>
<td>Annotating Plots</td>
<td>1-44</td>
</tr>
<tr>
<td>Annotation Object Properties</td>
<td>1-44</td>
</tr>
<tr>
<td>Specialized Plotting</td>
<td>1-44</td>
</tr>
<tr>
<td>Area, Bar, and Pie Plots</td>
<td>1-45</td>
</tr>
<tr>
<td>Contour Plots</td>
<td>1-45</td>
</tr>
<tr>
<td>Direction and Velocity Plots</td>
<td>1-45</td>
</tr>
<tr>
<td>Discrete Data Plots</td>
<td>1-45</td>
</tr>
<tr>
<td>Function Plots</td>
<td>1-45</td>
</tr>
<tr>
<td>Histograms</td>
<td>1-46</td>
</tr>
<tr>
<td>Polygons and Surfaces</td>
<td>1-46</td>
</tr>
<tr>
<td>Scatter/Bubble Plots</td>
<td>1-46</td>
</tr>
<tr>
<td>Animation</td>
<td>1-46</td>
</tr>
<tr>
<td>Bit-Mapped Images</td>
<td>1-47</td>
</tr>
<tr>
<td>Printing</td>
<td>1-47</td>
</tr>
<tr>
<td>Handle Graphics</td>
<td>1-47</td>
</tr>
<tr>
<td>Finding and Identifying Graphics Objects</td>
<td>1-48</td>
</tr>
<tr>
<td>Object Creation Functions</td>
<td>1-48</td>
</tr>
<tr>
<td>Plot Objects</td>
<td>1-48</td>
</tr>
<tr>
<td>Figure Windows</td>
<td>1-49</td>
</tr>
<tr>
<td>Axes Operations</td>
<td>1-49</td>
</tr>
<tr>
<td>Operating on Object Properties</td>
<td>1-49</td>
</tr>
<tr>
<td><strong>3-D Visualization</strong></td>
<td>1-50</td>
</tr>
<tr>
<td>Surface and Mesh Plots</td>
<td>1-50</td>
</tr>
<tr>
<td>Creating Surfaces and Meshes</td>
<td>1-50</td>
</tr>
<tr>
<td>Domain Generation</td>
<td>1-50</td>
</tr>
<tr>
<td>Color Operations</td>
<td>1-51</td>
</tr>
<tr>
<td>Colormaps</td>
<td>1-51</td>
</tr>
<tr>
<td>View Control</td>
<td>1-51</td>
</tr>
</tbody>
</table>
Controlling the Camera Viewpoint .......................... 1-52
Setting the Aspect Ratio and Axis Limits ....................... 1-52
Object Manipulation ........................................... 1-52
Selecting Region of Interest .................................. 1-52
Lighting .................................................................. 1-53
Transparency ....................................................... 1-53
Volume Visualization ............................................. 1-53

Creating Graphical User Interfaces ......................... 1-54
  Predefined Dialog Boxes .................................... 1-54
  Deploying User Interfaces .................................... 1-55
  Developing User Interfaces .................................... 1-55
    Working with Application Data ......................... 1-55
    Interactive User Input .................................... 1-55
  User Interface Objects ....................................... 1-55
  Finding Objects from Callbacks ............................ 1-55

Functions — Alphabetical List

  Stretch-to-Fill .................................................. 2-139
  Positioning the Axes .......................................... 2-140
  Setting Default Properties .................................. 2-142
  Properties That Control the X-, Y-, or Z-Axis .......... 2-170
  Specifying Colormaps ........................................ 2-387
  Supported Colormaps ......................................... 2-387
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td>Terminate MATLAB (same as quit)</td>
</tr>
<tr>
<td>finish</td>
<td>MATLAB termination M-file</td>
</tr>
<tr>
<td>genpath</td>
<td>Generate a path string</td>
</tr>
<tr>
<td>matlab</td>
<td>Start MATLAB (UNIX systems)</td>
</tr>
<tr>
<td>matlab</td>
<td>Start MATLAB (Windows systems)</td>
</tr>
<tr>
<td>matlabrc</td>
<td>MATLAB startup M-file for single user systems or administrators</td>
</tr>
<tr>
<td>prefdir</td>
<td>Return directory containing preferences, history, and layout files</td>
</tr>
<tr>
<td>preferences</td>
<td>Display Preferences dialog box for MATLAB and related products</td>
</tr>
<tr>
<td>quit</td>
<td>Terminate MATLAB</td>
</tr>
<tr>
<td>startup</td>
<td>MATLAB startup M-file for user-defined options</td>
</tr>
</tbody>
</table>
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**

- **assignin**  
  Assign value to workspace variable
- **clear**  
  Remove items from workspace, freeing up system memory
- **evalin**  
  Execute string containing MATLAB expression in a workspace
- **exist**  
  Check if variables or functions are defined
- **openvar**  
  Open workspace variable in Array Editor for graphical editing
- **pack**  
  Consolidate workspace memory
- **uiimport**  
  Open Import Wizard, the graphical user interface to import data
- **which**  
  Locate functions and files
- **who, whos**  
  List variables in the workspace
- **workspace**  
  Display Workspace browser, a tool for managing the workspace

**Search Path**

- **addpath**  
  Add directories to MATLAB search path
- **genpath**  
  Generate path string
- **partialpath**  
  Partial pathname
- **path**  
  View or change the MATLAB directory search path
- **path2rc**  
  Replaced by `savepath`
- **pathdef**  
  List of directories in the MATLAB search path
- **pathsep**  
  Return path separator for current platform
- **pathtool**  
  Open **Set Path** dialog box to view and change MATLAB path
- **restoredefaultpath**  
  Restore the default search path
- **rmpath**  
  Remove directories from MATLAB search path
- **savepath**  
  Save current MATLAB search path to `pathdef.m` file

**File Operations**

- **cd**  
  Change working directory
- **copyfile**  
  Copy file or directory
- **delete**  
  Delete files or graphics objects
- **dir**  
  Display directory listing
- **exist**  
  Check if variables or functions are defined
- **fileattrib**  
  Set or get attributes of file or directory
- **filebrowser**  
  Display Current Directory browser, a tool for viewing files
- **lookfor**  
  Search for specified keyword in all help entries
- **ls**  
  List directory on UNIX
- **matlabroot**  
  Return root directory of MATLAB installation
- **mkdir**  
  Make new directory
- **movefile**  
  Move file or directory
- **pwd**  
  Display current directory
- **recycle**  
  Set option to move deleted files to recycle folder
- **rehash**  
  Refresh function and file system path caches
- **rmdir**  
  Remove directory
type List file
web Open Web site or file in Web browser or Help browser
what List MATLAB specific files in current directory
which Locate functions and files

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
dbdowm Change local workspace context
dbqnit Quit debug mode
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
dbtype Edit or create M-file
keyboard Invoke the keyboard in an M-file

**Performance Improvement and Tuning Tools and Techniques**

dean Help for memory limitations
mllnt Check M-files for possible problems, and report results
mlint 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

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

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

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

### Linear Equations

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

### Eigenvalues and Singular Values

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
<td>Improve accuracy of computed eigenvalues</td>
</tr>
<tr>
<td>cdf2rdf</td>
<td>Convert complex diagonal form to real block diagonal form</td>
</tr>
<tr>
<td>condeig</td>
<td>Condition number with respect to eigenvalues</td>
</tr>
<tr>
<td>eig</td>
<td>Eigenvalues and eigenvectors</td>
</tr>
<tr>
<td>eigs</td>
<td>Eigenvalues and eigenvectors of sparse matrix</td>
</tr>
<tr>
<td>gsvd</td>
<td>Generalized singular value decomposition</td>
</tr>
<tr>
<td>hess</td>
<td>Hessenberg form of matrix</td>
</tr>
<tr>
<td>poly</td>
<td>Polynomial with specified roots</td>
</tr>
<tr>
<td>polyeig</td>
<td>Polynomial eigenvalue problem</td>
</tr>
<tr>
<td>qz</td>
<td>QZ factorization for generalized eigenvalues</td>
</tr>
</tbody>
</table>
### Functions — Categorical List

<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)”
<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>csd</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>
### Exponential

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp</code></td>
<td>Exponential</td>
</tr>
<tr>
<td><code>expm1</code></td>
<td>Exponential of x minus 1</td>
</tr>
<tr>
<td><code>log</code></td>
<td>Natural logarithm</td>
</tr>
<tr>
<td><code>log1p</code></td>
<td>Logarithm of 1+x</td>
</tr>
<tr>
<td><code>log2</code></td>
<td>Base 2 logarithm and dissect floating-point numbers into exponent and mantissa</td>
</tr>
<tr>
<td><code>log10</code></td>
<td>Common (base 10) logarithm</td>
</tr>
<tr>
<td><code>nextpow2</code></td>
<td>Next higher power of 2</td>
</tr>
<tr>
<td><code>pow2</code></td>
<td>Base 2 power and scale floating-point number</td>
</tr>
<tr>
<td><code>reallog</code></td>
<td>Natural logarithm for nonnegative real arrays</td>
</tr>
<tr>
<td><code>realpow</code></td>
<td>Array power for real-only output</td>
</tr>
<tr>
<td><code>realsqrt</code></td>
<td>Square root for nonnegative real arrays</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>Square root</td>
</tr>
<tr>
<td><code>nthroot</code></td>
<td>Real nth root</td>
</tr>
</tbody>
</table>

### Complex

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs</code></td>
<td>Absolute value</td>
</tr>
<tr>
<td><code>angle</code></td>
<td>Phase angle</td>
</tr>
<tr>
<td><code>complex</code></td>
<td>Construct complex data from real and imaginary parts</td>
</tr>
<tr>
<td><code>conj</code></td>
<td>Complex conjugate</td>
</tr>
<tr>
<td><code>cplxpair</code></td>
<td>Sort numbers into complex conjugate pairs</td>
</tr>
<tr>
<td><code>i</code></td>
<td>Imaginary unit</td>
</tr>
<tr>
<td><code>imag</code></td>
<td>Complex imaginary part</td>
</tr>
<tr>
<td><code>isreal</code></td>
<td>True for real array</td>
</tr>
<tr>
<td><code>j</code></td>
<td>Imaginary unit</td>
</tr>
<tr>
<td><code>real</code></td>
<td>Complex real part</td>
</tr>
<tr>
<td><code>sign</code></td>
<td>Signum</td>
</tr>
<tr>
<td><code>unwrap</code></td>
<td>Unwrap phase angle</td>
</tr>
</tbody>
</table>

### Rounding and Remainder

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fix</code></td>
<td>Round towards zero</td>
</tr>
<tr>
<td><code>floor</code></td>
<td>Round towards minus infinity</td>
</tr>
<tr>
<td><code>ceil</code></td>
<td>Round towards plus infinity</td>
</tr>
<tr>
<td><code>round</code></td>
<td>Round towards nearest integer</td>
</tr>
<tr>
<td><code>mod</code></td>
<td>Modulus after division</td>
</tr>
<tr>
<td><code>rem</code></td>
<td>Remainder after division</td>
</tr>
</tbody>
</table>
**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

corrcoef  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
decov     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
fft2      Two-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
decov     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 >= 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)

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

Delay Differential Equations

- **dde23**: Solve delay differential equations with constant delays
- **ddeget**: Get DDE options parameters
- **ddeset**: Create/alter DDE options structure
- **deval**: Evaluate solution of differential equation problem

Boundary Value Problems

- **bvp4c**: Solve boundary value problems for ODEs
- **bvpget**: Get BVP options parameters
- **bvpset**: Create/alter BVP options structure
- **deval**: Evaluate solution of differential equation problem

Partial Differential Equations

- **pdepe**: Solve initial-boundary value problems for parabolic-elliptic PDEs
- **pdeval**: Evaluates by interpolation solution computed by pdepe

Optimization

- **fminbnd**: Scalar bounded nonlinear function minimization
- **fminsearch**: Multidimensional unconstrained nonlinear minimization, by Nelder-Mead direct search method
- **fzero**: Scalar nonlinear zero finding
- **lsqnonneg**: Linear least squares with nonnegativity constraints
- **optimset**: Create or alter optimization options structure
- **optimget**: Get optimization parameters from options structure
### 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spdiags</td>
<td>Sparse matrix formed from diagonals</td>
</tr>
<tr>
<td>speye</td>
<td>Sparse identity matrix</td>
</tr>
<tr>
<td>sprand</td>
<td>Sparse uniformly distributed random matrix</td>
</tr>
<tr>
<td>sprandn</td>
<td>Sparse normally distributed random matrix</td>
</tr>
<tr>
<td>sprandsym</td>
<td>Sparse random symmetric matrix</td>
</tr>
</tbody>
</table>

### Full to Sparse Conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>Find indices of nonzero elements</td>
</tr>
<tr>
<td>full</td>
<td>Convert sparse matrix to full matrix</td>
</tr>
<tr>
<td>sparse</td>
<td>Create sparse matrix</td>
</tr>
<tr>
<td>spconvert</td>
<td>Import from sparse matrix external format</td>
</tr>
</tbody>
</table>

### Working with Sparse Matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>issparse</td>
<td>True for sparse matrix</td>
</tr>
<tr>
<td>nnz</td>
<td>Number of nonzero matrix elements</td>
</tr>
<tr>
<td>nonzeros</td>
<td>Nonzero matrix elements</td>
</tr>
<tr>
<td>nzmax</td>
<td>Amount of storage allocated for nonzero matrix elements</td>
</tr>
<tr>
<td>spalloc</td>
<td>Allocate space for sparse matrix</td>
</tr>
<tr>
<td>spfun</td>
<td>Apply function to nonzero matrix elements</td>
</tr>
<tr>
<td>spones</td>
<td>Replace nonzero sparse matrix elements with ones</td>
</tr>
<tr>
<td>spparms</td>
<td>Set parameters for sparse matrix routines</td>
</tr>
<tr>
<td>spy</td>
<td>Visualize sparsity pattern</td>
</tr>
</tbody>
</table>

### Reordering Algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colamd</td>
<td>Column approximate minimum degree permutation</td>
</tr>
<tr>
<td>colmmd</td>
<td>Column minimum degree permutation</td>
</tr>
<tr>
<td>colperm</td>
<td>Column permutation</td>
</tr>
<tr>
<td>dmperm</td>
<td>Dulmage-Mendelsohn permutation</td>
</tr>
<tr>
<td>randperm</td>
<td>Random permutation</td>
</tr>
<tr>
<td>symamd</td>
<td>Symmetric approximate minimum degree permutation</td>
</tr>
<tr>
<td>symmmd</td>
<td>Symmetric minimum degree permutation</td>
</tr>
<tr>
<td>symrcm</td>
<td>Symmetric reverse Cuthill-McKee permutation</td>
</tr>
</tbody>
</table>

### Linear Algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholinc</td>
<td>Incomplete Cholesky factorization</td>
</tr>
<tr>
<td>condest</td>
<td>1-norm condition number estimate</td>
</tr>
<tr>
<td>eigs</td>
<td>Eigenvalues and eigenvectors of sparse matrix</td>
</tr>
<tr>
<td>luinc</td>
<td>Incomplete LU factorization</td>
</tr>
<tr>
<td>normest</td>
<td>Estimate matrix 2-norm</td>
</tr>
<tr>
<td>sprank</td>
<td>Structural rank</td>
</tr>
<tr>
<td>svds</td>
<td>Singular values and vectors of sparse matrix</td>
</tr>
</tbody>
</table>
Linear Equations (Iterative Methods)

bicg
bicgstab
conjugate gradients method
BiConjugate Gradients Stabilized method
Conjugate Gradients Squared method
Generalized Minimum Residual method
LSQR implementation of Conjugate Gradients on Normal Equations
Minimum Residual method
Preconditioned Conjugate Gradients method
Quasi-Minimal Residual method
Form least squares augmented system
Symmetric LQ method

Tree Operations

etree
etreeplot
elimination tree
Plot elimination tree
Plot graph, as in “graph theory”
Symbolic factorization analysis
Lay out tree or forest
Plot picture of tree

Math Constants

eps
floating-point relative accuracy
i
Imaginary unit
Inf
Infinity, ∞
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, π
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”
### 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>isreal</td>
<td>Determine if all array elements are real numbers</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

- **strings** Describes MATLAB string handling

#### 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>
</tbody>
</table>
strjust Justify character array
strread Read formatted data from string
strrep String search and replace
strtrim Remove leading and trailing whitespace from string
strvcat Vertical concatenation of strings
upper Convert string to upper case

Comparing and Searching Strings

class Return object’s class name (e.g., char)
findstr Find string within another, longer string
isa Determine if item is object of given class (e.g., char)
iscellstr Determine if item is cell array of strings
ischar Determine if item is character array
isletter Detect array elements that are letters of the alphabet
isscalar True for scalars (1-by-1 matrices)
isascii Detect elements that are ASCII white spaces
isstrprop Determine content of each element of string
isvector True for vectors (1-by-N or N-by-1 matrices)
regexp Match regular expression
regexpi Match regular expression, ignoring case
regexprep Replace string using regular expression
strncmp Compare first n characters of strings
strncmpi Compare first n characters of strings, ignoring case
strtok First token in string

Evaluating String Expressions

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

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

**Cell Arrays**

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

### Numeric

<table>
<thead>
<tr>
<th>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>Function</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>Function</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>Function</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>
### Determine Data Type

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is*</td>
<td>Detect state</td>
</tr>
<tr>
<td>isa</td>
<td>Determine if item is object of given class</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>ischar</td>
<td>Determine if item is character array</td>
</tr>
<tr>
<td>isfield</td>
<td>Determine if item is character array</td>
</tr>
<tr>
<td>isfloat</td>
<td>True for floating-point arrays</td>
</tr>
<tr>
<td>isinteger</td>
<td>True for integer arrays</td>
</tr>
<tr>
<td>isjava</td>
<td>Determine if item is Java object</td>
</tr>
<tr>
<td>islogical</td>
<td>Determine if item is logical array</td>
</tr>
<tr>
<td>isnumeric</td>
<td>Determine if item is numeric array</td>
</tr>
<tr>
<td>isobject</td>
<td>Determine if item is MATLAB OOPs object</td>
</tr>
<tr>
<td>isreal</td>
<td>Determine if all array elements are real numbers</td>
</tr>
<tr>
<td>isstruct</td>
<td>Determine if item is MATLAB structure array</td>
</tr>
</tbody>
</table>

### Arrays

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

### Array Operations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>Array constructor</td>
</tr>
<tr>
<td></td>
<td>Array row element separator</td>
</tr>
<tr>
<td>,</td>
<td>Array column element separator</td>
</tr>
<tr>
<td>;</td>
<td>Specify range of array elements</td>
</tr>
<tr>
<td>end</td>
<td>Indicate last index of array</td>
</tr>
<tr>
<td>+</td>
<td>Addition or unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or unary minus</td>
</tr>
<tr>
<td>.*</td>
<td>Array multiplication</td>
</tr>
<tr>
<td>./</td>
<td>Array right division</td>
</tr>
<tr>
<td>.\</td>
<td>Array left division</td>
</tr>
<tr>
<td>.^</td>
<td>Array power</td>
</tr>
<tr>
<td>.’</td>
<td>Array (nonconjugated) transpose</td>
</tr>
</tbody>
</table>
## Basic Array Information

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

## Array Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>:</code></td>
<td>Specify range of array elements</td>
</tr>
<tr>
<td><code>blkdiag</code></td>
<td>Construct block diagonal matrix from input arguments</td>
</tr>
<tr>
<td><code>cat</code></td>
<td>Concatenate arrays</td>
</tr>
<tr>
<td><code>circshift</code></td>
<td>Shift array circularly</td>
</tr>
<tr>
<td><code>find</code></td>
<td>Find indices and values of nonzero elements</td>
</tr>
<tr>
<td><code>fliplr</code></td>
<td>Flip matrices left-right</td>
</tr>
<tr>
<td><code>flipud</code></td>
<td>Flip matrices up-down</td>
</tr>
<tr>
<td><code>flipdim</code></td>
<td>Flip array along specified dimension</td>
</tr>
<tr>
<td><code>horzcat</code></td>
<td>Horizontal concatenation</td>
</tr>
<tr>
<td><code>ind2sub</code></td>
<td>Subscripts from linear index</td>
</tr>
<tr>
<td><code>ipermute</code></td>
<td>Inverse permute dimensions of multidimensional array</td>
</tr>
<tr>
<td><code>permute</code></td>
<td>Rearrange dimensions of multidimensional array</td>
</tr>
<tr>
<td><code>repmat</code></td>
<td>Replicate and tile array</td>
</tr>
<tr>
<td><code>reshape</code></td>
<td>Reshape array</td>
</tr>
<tr>
<td><code>rot90</code></td>
<td>Rotate matrix 90 degrees</td>
</tr>
<tr>
<td><code>shiftdim</code></td>
<td>Shift dimensions</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Sort array elements in ascending or descending order</td>
</tr>
<tr>
<td><code>sortrows</code></td>
<td>Sort rows in ascending order</td>
</tr>
<tr>
<td><code>squeeze</code></td>
<td>Remove singleton dimensions</td>
</tr>
<tr>
<td><code>sub2ind</code></td>
<td>Single index from subscripts</td>
</tr>
<tr>
<td><code>vertcat</code></td>
<td>Horizontal concatenation</td>
</tr>
</tbody>
</table>
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
namelengthmax 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

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

### Timer Functions

- **delete**: Delete timer object from memory
- **disp**: Display information about timer object
- **get**: Retrieve information about timer object properties
- **isvalid**: Determine if timer object is valid
- **set**: Display or set timer object properties
- **start**: Start a timer
- **startat**: Start a timer at a specific timer
- **stop**: Stop a timer
- **timer**: Create a timer object
- **timerfind**: Return an array of all visible timer objects in memory
- **timerfindall**: Return an array of all timer objects in memory
- **wait**: Block command line until timer completes

### Variables and Functions in Memory

- **assignin**: Assign value to workspace variable
- **genvarname**: Construct valid variable name from string
- **global**: Define global variables
- **inmem**: Return names of functions in memory
- **isglobal**: Determine if item is global variable
- **mislocked**: True if M-file cannot be cleared
- **mlock**: Prevent clearing M-file from memory
- **munlock**: Allow clearing M-file from memory
- **namelengthmax**: Return maximum identifier length
- **pack**: Consolidate workspace memory
- **persistent**: Define persistent variable
- **rehash**: Refresh function and file system caches
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
methodview 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
defun     List Java classes used by M-file
depfun    List Java classes used by M-file
eexist    Determine if item is Java class
fieldname 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
javaddpath 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
methodview 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
error    Display error message
ferror   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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbmex</code></td>
<td>Enable MEX-file debugging</td>
</tr>
<tr>
<td><code>inmem</code></td>
<td>Return names of currently loaded MEX-files</td>
</tr>
<tr>
<td><code>mex</code></td>
<td>Compile MEX-function from C or Fortran source code</td>
</tr>
<tr>
<td><code>mexext</code></td>
<td>Return MEX-filename extension</td>
</tr>
</tbody>
</table>
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
uimport     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)
fgetl       Return next line of file as string without line terminator(s)
fopen       Open file or obtain information about open files
fprintf      Write formatted data to file
fwrite       Write binary data to file
fread        Read binary data from file
frewind      Rewind open file
fscanf       Read formatted data from file
fseek        Set file position indicator
ftell        Get file position indicator

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

Basic Plots and Graphs
- Linear line plots, log and semilog plots

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

### 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

- **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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>Area plot</td>
</tr>
<tr>
<td>bar</td>
<td>Vertical bar chart</td>
</tr>
<tr>
<td>barh</td>
<td>Horizontal bar chart</td>
</tr>
<tr>
<td>bar3</td>
<td>Vertical 3-D bar chart</td>
</tr>
<tr>
<td>bar3h</td>
<td>Horizontal 3-D bar chart</td>
</tr>
<tr>
<td>pareto</td>
<td>Pareto char</td>
</tr>
<tr>
<td>pie</td>
<td>Pie plot</td>
</tr>
<tr>
<td>pie3</td>
<td>3-D pie plot</td>
</tr>
</tbody>
</table>

## Contour Plots

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contour</td>
<td>Contour (level curves) plot</td>
</tr>
<tr>
<td>contour3</td>
<td>3-D contour plot</td>
</tr>
<tr>
<td>contourc</td>
<td>Contour computation</td>
</tr>
<tr>
<td>contourf</td>
<td>Filled contour plot</td>
</tr>
<tr>
<td>ezcontour</td>
<td>Easy to use contour plotter</td>
</tr>
<tr>
<td>ezcontourf</td>
<td>Easy to use filled contour plotter</td>
</tr>
</tbody>
</table>

## Direction and Velocity Plots

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comet</td>
<td>Comet plot</td>
</tr>
<tr>
<td>comet3</td>
<td>3-D comet plot</td>
</tr>
<tr>
<td>compass</td>
<td>Compass plot</td>
</tr>
<tr>
<td>feather</td>
<td>Feather plot</td>
</tr>
<tr>
<td>quiver</td>
<td>Quiver (or velocity) plot</td>
</tr>
<tr>
<td>quiver3</td>
<td>3-D quiver (or velocity) plot</td>
</tr>
</tbody>
</table>

## Discrete Data Plots

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stem</td>
<td>Plot discrete sequence data</td>
</tr>
<tr>
<td>stem3</td>
<td>Plot discrete surface data</td>
</tr>
<tr>
<td>stairs</td>
<td>Stairstep graph</td>
</tr>
</tbody>
</table>

## Function Plots

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ezcontour</td>
<td>Easy to use contour plotter</td>
</tr>
<tr>
<td>ezcontourf</td>
<td>Easy to use filled contour plotter</td>
</tr>
<tr>
<td>ezmesh</td>
<td>Easy to use 3-D mesh plotter</td>
</tr>
<tr>
<td>ezmeshc</td>
<td>Easy to use combination mesh/contour plotter</td>
</tr>
<tr>
<td>ezplot</td>
<td>Easy to use function plotter</td>
</tr>
<tr>
<td>ezplot3</td>
<td>Easy to use 3-D parametric curve plotter</td>
</tr>
<tr>
<td>ezpolar</td>
<td>Easy to use polar coordinate plotter</td>
</tr>
<tr>
<td>ezsurf</td>
<td>Easy to use 3-D colored surface plotter</td>
</tr>
<tr>
<td>ezsurfc</td>
<td>Easy to use combination surface/contour plotter</td>
</tr>
<tr>
<td>fplot</td>
<td>Plot a function</td>
</tr>
</tbody>
</table>
### Histograms

- **hist**: Plot histograms
- **histc**: Histogram count
- **rose**: Plot rose or angle histogram

### Polygons and Surfaces

- **convhull**: Convex hull
- **cylinder**: Generate cylinder
- ** delaunay**: Delaunay triangulation
- ** dsearch**: Search Delaunay triangulation for nearest point
- ** ellipsoid**: Generate ellipsoid
- ** fill**: Draw filled 2-D polygons
- ** fill3**: Draw filled 3-D polygons in 3-space
- ** inpoly**: True for points inside a polygonal region
- ** pcolor**: Pseudocolor (checkerboard) plot
- ** polyarea**: Area of polygon
- ** ribbon**: Ribbon plot
- ** slice**: Volumetric slice plot
- ** sphere**: Generate sphere
- ** tsearch**: Search for enclosing Delaunay triangle
- ** voronoi**: Voronoi diagram
- ** waterfall**: Waterfall plot

### Scatter/Bubble Plots

- ** plotmatrix**: Scatter plot matrix
- ** scatter**: Scatter plot
- ** scatter3**: 3-D scatter plot

### Animation

- ** frame2im**: Convert movie frame to indexed image
- ** getframe**: Capture movie frame
- ** im2frame**: Convert image to movie frame
- ** movie**: Play recorded movie frames
- ** noanimate**: Change EraseMode of all objects to normal
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
errorbarseries  Property list
lineseries  Property list
quivergroup  Property list
scattergroup  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
hgsload 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
- meshc
- mesh
- peaks
- surf
- surface
- surfc
- surfl
- tetramesh
- trimesh
- triplot
- trisurf

Domain Generation
- griddata
- meshgrid

Data gridding and surface fitting
Generation of X and Y arrays for 3-D plots
3-D Visualization

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
- **surfnorm**: 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

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

Setting the Aspect Ratio and Axis Limits

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>daspect</td>
<td>Set or get data aspect ratio</td>
</tr>
<tr>
<td>pbaspect</td>
<td>Set or get plot box aspect ratio</td>
</tr>
<tr>
<td>xlim</td>
<td>Set or get the current x-axis limits</td>
</tr>
<tr>
<td>ylim</td>
<td>Set or get the current y-axis limits</td>
</tr>
<tr>
<td>zlim</td>
<td>Set or get the current z-axis limits</td>
</tr>
</tbody>
</table>

Object Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pan</td>
<td>Turns panning on or off</td>
</tr>
<tr>
<td>reset</td>
<td>Reset axis or figure</td>
</tr>
<tr>
<td>rotate</td>
<td>Rotate objects about specified origin and direction</td>
</tr>
<tr>
<td>rotate3d</td>
<td>Interactively rotate the view of a 3-D plot</td>
</tr>
<tr>
<td>selectmoveresize</td>
<td>Interactively select, move, or resize objects</td>
</tr>
<tr>
<td>zoom</td>
<td>Zoom in and out on a 2-D plot</td>
</tr>
</tbody>
</table>

Selecting Region of Interest

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dragrect</td>
<td>Drag XOR rectangles with mouse</td>
</tr>
<tr>
<td>rbbox</td>
<td>Rubberband box</td>
</tr>
</tbody>
</table>
**Lighting**

camlight  
light  
lightangle  
lighting  
material

**Transparency**

alpha  
alphamap  
alim

**Volume Visualization**

coneplot  
contourslice  
curl  
divergence  
flow  
interplstreamspeed  
isocaps  
isocolors  
isonormals  
isosurface  
reducepatch  
reducevolume  
shrinkfaces  
slice  
smooth3  
stream2  
stream3  
streamline  
streamparticles  
streamribbon  
streamslices  
streamtube  
surf2patch  
subvolume  
volumebounds
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
- uisecolor: Set ColorSpec using dialog box
- uisefont: 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

guidata Store or retrieve application data
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/button press over figure

User Interface Objects

menu Generate menu of choices for user input
uibuttongroup Create component to exclusively manage radiobuttons and togglebuttons
uicontextmenu Create context menu
uicontrol Create user interface control
uimenu Create user interface menu
uipanel Create panel container object
uipushtool Create toolbar push button
uittoggletool Create toolbar toggle button
uistoolbar 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 — Categorical List
Functions — Alphabetical List
Arithmetic Operators + - * / \ ^ '

Purpose
Matrix and array arithmetic

Syntax
A+B
A-B
A+B A.*B
A/B A./B
A\B A\B
A.^B A.^B
A' A.'

Description
MATLAB has two different types of arithmetic operations. Matrix arithmetic operations are defined by the rules of linear algebra. Array arithmetic operations are carried out element by element, and can be used with multidimensional arrays. The period character (.) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs .+ and .- are not used.

+  Addition or unary plus. A+B adds A and B. A and B must have the same size, unless one is a scalar. A scalar can be added to a matrix of any size.

-  Subtraction or unary minus. A-B subtracts B from A. A and B must have the same size, unless one is a scalar. A scalar can be subtracted from a matrix of any size.

*  Matrix multiplication. C = A*B is the linear algebraic product of the matrices A and B. More precisely,

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

For nonscalar A and B, the number of columns of A must equal the number of rows of B. A scalar can multiply a matrix of any size.

.* Array multiplication. A.*B is the element-by-element product of the arrays A and B. A and B must have the same size, unless one of them is a scalar.
Arithmetic Operators + - * / \ ^ '

/  Slash or matrix right division. B/A is roughly the same as B*inv(A).
More precisely, B/A = (A\B')'. See the reference page for mrdivide for
more information.

./ Array right division. A./B is the matrix with elements A(i,j)/B(i,j).
A and B must have the same size, unless one of them is a scalar.

\ Backslash or matrix left division. If A is a square matrix, A\B is roughly
the same as inv(A)*B, except it is computed in a different way. If A is an
n-by-n matrix and B is a column vector with n components, or a matrix
with several such columns, then X = A\B is the solution to the equation
AX = B computed by Gaussian elimination. A warning message is
displayed if A is badly scaled or nearly singular. See the reference page
for mldivide for more information.

If A is an m-by-n matrix with m ~= n and B is a column vector with m
components, or a matrix with several such columns, then X = A\B is the
solution in the least squares sense to the under- or overdetermined
system of equations AX = B. The effective rank, k, of A is determined
from the QR decomposition with pivoting (see “Algorithm” on
page 2-701 for details). A solution X is computed that has at most k
nonzero components per column. If k < n, this is usually not the same
solution as pinv(A)*B, which is the least squares solution with the
smallest norm ||X||.

\ Array left division. A.\B is the matrix with elements B(i,j)/A(i,j). A
and B must have the same size, unless one of them is a scalar.

^ Matrix power. X^p is X to the power p, if p is a scalar. If p is an integer,
the power is computed by repeated squaring. If the integer is negative,
X is inverted first. For other values of p, the calculation involves
eigenvalues and eigenvectors, such that if [V,D] = eig(X), then
X^p = V*D.^p/V.

If x is a scalar and P is a matrix, x^P is x raised to the matrix power P
using eigenvalues and eigenvectors. X^P, where X and P are both
matrices, is an error.

^ Array power. A.^B is the matrix with elements A(i,j) to the B(i,j)
power. A and B must have the same size, unless one of them is a scalar.
Arithmetic Operators + - * / \ ^ '

This section describes the arithmetic operators' support for data types other than double.

**Data Type single**
You can apply any of the arithmetic operators to arrays of type single and MATLAB returns an answer of type single. You can also combine an array of type double with an array of type single, and the result has type single.

**Integer Data Types**
You can apply most of the arithmetic operators to real arrays of the following integer data types:

- int8 and uint8
- int16 and uint16
- int32 and uint32

All operands must have the same integer data type and MATLAB returns an answer of that type.

**Note** The arithmetic operators do not support operations on the data types int64 or uint64. Except for the unary operators +A and A.', the arithmetic operators do not support operations on complex arrays of any integer data type.

For example,
```
x = int8(3) + int8(4);
class(x)
```
```
ans =
```
The following table lists the binary arithmetic operators that you can apply to arrays of the same integer data type. In the table, A and B are arrays of the same integer data type and c is a scalar of type double or the same type as A and B.

### Operation Support when A and B Have Same Integer Type

<table>
<thead>
<tr>
<th>Operation</th>
<th>Support when A and B Have Same Integer Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+A, -A</td>
<td>Yes</td>
</tr>
<tr>
<td>A+B, A+c, c+B</td>
<td>Yes</td>
</tr>
<tr>
<td>A-B, A-c, c-B</td>
<td>Yes</td>
</tr>
<tr>
<td>A.*B</td>
<td>Yes</td>
</tr>
<tr>
<td>A<em>c, c</em>B</td>
<td>Yes</td>
</tr>
<tr>
<td>A*B</td>
<td>No</td>
</tr>
<tr>
<td>A/c, c/B</td>
<td>Yes</td>
</tr>
<tr>
<td>A\ B, A./B</td>
<td>Yes</td>
</tr>
<tr>
<td>A\ B, A/B</td>
<td>No</td>
</tr>
<tr>
<td>A.^B</td>
<td>Yes, if B has nonnegative integer values.</td>
</tr>
<tr>
<td>c^k</td>
<td>Yes, for a scalar c and a nonnegative scalar integer k, which have the same integer data type or one of which has type double</td>
</tr>
</tbody>
</table>
| A

### Combining Integer Data Types with Type Double

For the operations that support integer data types, you can combine a scalar or array of an integer data type with a scalar, but not an array, of type double and the result has the same integer data type as the input of integer type. For example,

\[ y = 5 + \text{int32}(7); \]
class(y)
ans =
int32

However, you cannot combine an array of an integer data type with either of the following:

- A scalar or array of a different integer data type
- A scalar or array of type single

Nondouble Data Types, in the online MATLAB documentation, provides more information about operations on nondouble data types.

**Remarks**

The arithmetic operators have M-file function equivalents, as shown:

- **Binary addition**
  
  A + B  
  plus(A,B)

- **Unary plus**
  
  +A  
  uplus(A)

- **Binary subtraction**
  
  A - B  
  minus(A,B)

- **Unary minus**
  
  -A  
  uminus(A)

- **Matrix multiplication**
  
  A * B  
  mtimes(A,B)

- **Arraywise multiplication**
  
  A .* B  
  times(A,B)

- **Matrix right division**
  
  A / B  
  mrdivide(A,B)

- **Arraywise right division**
  
  A ./ B  
  rdivide(A,B)

- **Matrix left division**
  
  A \ B  
  mldivide(A,B)

- **Arraywise left division**
  
  A .\ B  
  ldivide(A,B)

- **Matrix power**
  
  A ^ B  
  mpower(A,B)

- **Arraywise power**
  
  A .^ B  
  power(A,B)

- **Complex transpose**
  
  A'  
  ctranspose(A)

- **Matrix transpose**
  
  A .'  
  transpose(A)
**Arithmetic Operators + - * / \ ^'**

**Note** For some toolboxes, the arithmetic operators are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given operator, type `help` followed by the operator name. For example, type `help plus`. The toolboxes that overload `plus (+)` are listed. For information about using the operator in that toolbox, see the documentation for the toolbox.

**Examples**

Here are two vectors, and the results of various matrix and array operations on them, printed with `format rat`.

<table>
<thead>
<tr>
<th>Matrix Operations</th>
<th>Array Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>x'</td>
<td>y'</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>x + y</td>
<td>x - y</td>
</tr>
<tr>
<td>5</td>
<td>-3</td>
</tr>
<tr>
<td>7</td>
<td>-3</td>
</tr>
<tr>
<td>9</td>
<td>-3</td>
</tr>
<tr>
<td>x + 2</td>
<td>x - 2</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>x * y</td>
<td>x.*y</td>
</tr>
<tr>
<td>Error</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>18</td>
</tr>
<tr>
<td>x' * y</td>
<td>x'.*y</td>
</tr>
<tr>
<td>32</td>
<td>Error</td>
</tr>
<tr>
<td>x*y'</td>
<td>x*y'</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Error</td>
</tr>
<tr>
<td>x * 2</td>
<td>x.*2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
## Arithmetic Operators + - * / \ ^ ' 

<table>
<thead>
<tr>
<th>Matrix Operations</th>
<th>Array Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x\y</code></td>
<td>16/7</td>
</tr>
<tr>
<td></td>
<td><code>x./y</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>2\x</code></td>
<td>1/2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>x/y</code></td>
<td>0 0 0 1/6</td>
</tr>
<tr>
<td></td>
<td>0 0 1/3</td>
</tr>
<tr>
<td></td>
<td>0 0 1/2</td>
</tr>
<tr>
<td></td>
<td><code>x./y</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>x/2</code></td>
<td>1/2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>x^y</code></td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td><code>x.^y</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>x^2</code></td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td><code>x.^2</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>2^x</code></td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td><code>2.^x</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(x+i*y)'</code></td>
<td>1 - 4i 2 - 5i 3 - 6i</td>
</tr>
<tr>
<td><code>(x+i*y).'</code></td>
<td>1 + 4i 2 + 5i 3 + 6i</td>
</tr>
</tbody>
</table>

### Diagnostics

- From matrix division, if a square $A$ is singular, 
  Warning: Matrix is singular to working precision.

- From elementwise division, if the divisor has zero elements, 
  Warning: Divide by zero.

Matrix division and elementwise division can produce NaNs or Infs where appropriate.
Arithmetic Operators + - * / \ ^ '

- If the inverse was found, but is not reliable,
  Warning: Matrix is close to singular or badly scaled.
  Results may be inaccurate. RCOND = xxx

- From matrix division, if a nonsquare A is rank deficient,
  Warning: Rank deficient, rank = xxx tol = xxx

See Also
mldivide, mrdivide, chol, det, inv, lu, orth, permute, ipermute, qr, rref
Arithmetic Operators + - * / \ ^ '  

References


Relational Operators <  >  <=  >=  ==  ~=

**Purpose**
Relational operations

**Syntax**

- A < B
- A > B
- A <= B
- A >= B
- A == B
- A ~= B

**Description**
The relational operators are <, >, <=, >=, ==, and ~=. Relational operators perform element-by-element comparisons between two arrays. They return a logical array of the same size, with elements set to true (1) where the relation is true, and elements set to false (0) where it is not.

The operators <, >, <=, and >= use only the real part of their operands for the comparison. The operators == and ~= test real and imaginary parts.

To test if two strings are equivalent, use `strcmp`, which allows vectors of dissimilar length to be compared.

**Note**
For some toolboxes, the relational operators are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given operator, type `help` followed by the operator name. For example, type `help lt`. The toolboxes that overload `lt` (<) are listed. For information about using the operator in that toolbox, see the documentation for the toolbox.

**Examples**
If one of the operands is a scalar and the other a matrix, the scalar expands to the size of the matrix. For example, the two pairs of statements

```matlab
X = 5; X >= [1 2 3; 4 5 6; 7 8 10]
X = 5*ones(3,3); X >= [1 2 3; 4 5 6; 7 8 10]
```
produce the same result:

```
ans =

1 1 1
```
Relational Operators <  >  <=  >=  ==  ~=

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

See Also  

all, any, find, strcmp

Elementwise Logical Operators, &, |, Short-Circuit Logical Operators, &&, ||, -
Logical Operators: Elementwise & | ~

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Elementwise logical operations on arrays</th>
</tr>
</thead>
</table>
| Syntax  | A & B  
A | B  
~A |
| Description | The symbols &, |, and – are the logical array operators AND, OR, and NOT. They work element by element on arrays, with 0 representing logical false, and anything nonzero representing logical true. The logical operators return a logical array with elements set to true (1) or false (0), as appropriate. The & operator does a logical AND, the | operator does a logical OR, and ~A complements the elements of A. The function xor(A,B) implements the exclusive OR operation. The truth table for these operators and functions is shown below.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>and</th>
<th>or</th>
<th>not</th>
<th>xor</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &amp; B</td>
<td>A &amp; B</td>
<td>A</td>
<td>B</td>
<td>-A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The precedence for the logical operators with respect to each other is

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>NOT</td>
<td>Highest</td>
</tr>
<tr>
<td>&amp;</td>
<td>Elementwise AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Elementwise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Short-circuit AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Logical Operators: Elementwise & | ~

Remarks MATLAB always gives the & operator precedence over the | operator. Although MATLAB typically evaluates expressions from left to right, the expression a | b & c is evaluated as a | (b & c). It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of & and |.

These logical operators have M-file function equivalents, as shown.

<table>
<thead>
<tr>
<th>Logical Operation</th>
<th>Equivalent Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &amp; B</td>
<td>and(A,B)</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>¬A</td>
<td>not(A)</td>
</tr>
</tbody>
</table>

Examples This example shows the logical OR of the elements in the vector u with the corresponding elements in the vector v:

```matlab
t = [0 0 1 1 0 1];
v = [0 1 1 0 0 1];
u | v
```

```matlab
ans =
0 1 1 1 0 1
```

See Also all, any, find, logical, xor, true, false

Logical operators, short-circuit, &&, ||

Relational operators <, <=, >, >=, ==, ~=
Logical Operators: Short-circuit && ||

Purpose
Logical operations, with short-circuiting capability

Syntax
A && B
A || B

Description
The symbols && and || are the logical AND and OR operators used to evaluate logical expressions. Use && and || in the evaluation of compound expressions of the form

expression_1 && expression_2

where expression_1 and expression_2 each evaluate to a scalar logical result.

The && and || operators support short-circuiting. This means that the second operand is evaluated only when the result is not fully determined by the first operand. See “Short-Circuit Operators” in the MATLAB documentation for a discussion on short-circuiting with && and ||.

Note
Always use the && and || operators when short-circuiting is required. Using the elementwise operators (& and |) for short-circuiting can yield unexpected results.

Examples
In the following statement, it doesn’t make sense to evaluate the relation on the right if the divisor, b, is zero. The test on the left is put in to avoid generating a warning under these circumstances:

x = (b ~= 0) && (a/b > 18.5)

By definition, if any operands of an AND expression are false, the entire expression must be false. So, if (b ~= 0) evaluates to false, MATLAB assumes the entire expression to be false and terminates its evaluation of the expression early. This avoids the warning that would be generated if MATLAB were to evaluate the operand on the right.
Logical Operators: Short-circuit && ||

See Also
all, any, find, logical, xor, true, false
Logical operators, elementwise, &, |, ~
Relational operators <, <=, >, >=, ==, ~=
Special Characters [ ] ( ) {} = ' . ... , ; : % ! @

Purpose

Special characters

Syntax

[ ] ( ) {} = ' . ... , ; : % ! @

Description

[ ]  Brackets are used to form vectors and matrices. [6.9 9.64 sqrt(-1)] is a vector with three elements separated by blanks. [6.9, 9.64, i] is the same thing. [1+j 2-j 3] and [1+j 2 -j 3] are not the same. The first has three elements, the second has five.

[11 12 13; 21 22 23] is a 2-by-3 matrix. The semicolon ends the first row.

Vectors and matrices can be used inside [ ] brackets. [A B;C] is allowed if the number of rows of A equals the number of rows of B and the number of columns of A plus the number of columns of B equals the number of columns of C. This rule generalizes in a hopefully obvious way to allow fairly complicated constructions.

A = [ ] stores an empty matrix in A. A(m,:) = [ ] deletes row m of A.
A(:,n) = [ ] deletes column n of A. A(n) = [ ] reshapes A into a column vector and deletes the third element.

[A1,A2,A3...] = function assigns function output to multiple variables.
For the use of [ and ] on the left of an “=” in multiple assignment statements, see lu, eig, svd, and so on.

{ }  Curly braces are used in cell array assignment statements. For example, A(2,1) = {%1 2 3; 4 5 6}, or A{2,2} = (’str’). See help paren for more information about { }.
Parentheses are used to indicate precedence in arithmetic expressions in the usual way. They are used to enclose arguments of functions in the usual way. They are also used to enclose subscripts of vectors and matrices in a manner somewhat more general than usual. If \( X \) and \( V \) are vectors, then \( X(V) \) is \([X(V(1)), X(V(2)), \ldots, X(V(n))]\). The components of \( V \) must be integers to be used as subscripts. An error occurs if any such subscript is less than 1 or greater than the size of \( X \). Some examples are

- \( X(3) \) is the third element of \( X \).
- \( X([1 \ 2 \ 3]) \) is the first three elements of \( X \).

See `help paren` for more information about ( ).

If \( X \) has \( n \) components, \( X(n:1:1) \) reverses them. The same indirect subcripting works in matrices. If \( V \) has \( m \) components and \( W \) has \( n \) components, then \( A(V,W) \) is the \( m \)-by-\( n \) matrix formed from the elements of \( A \) whose subscripts are the elements of \( V \) and \( W \). For example, \( A([1\ 5],:) = A([5\ 1],:) \) interchanges rows 1 and 5 of \( A \).

= Used in assignment statements. \( B = A \) stores the elements of \( A \) in \( B \).

== is the relational equals operator. See the Relational Operators page.

' Matrix transpose. \( X' \) is the complex conjugate transpose of \( X \). \( X' \) is the nonconjugate transpose.

Quotation mark. 'any text' is a vector whose components are the ASCII codes for the characters. A quotation mark within the text is indicated by two quotation marks.

. Decimal point. 314/100, 3.14, and .314e1 are all the same.
Element-by-element operations. These are obtained using .*, .^, ./, or .\. See the Arithmetic Operators page.

. Field access. \( A.(field) \) and \( A(i).field \), when \( A \) is a structure, access the contents of \( field \).

.. Parent directory. See `cd`.
Remarks

Some uses of special characters have M-file function equivalents, as shown:

- **Horizontal concatenation**
  
  \[ [A,B,C\ldots] \quad \text{horzcat}(A,B,C\ldots) \]

- **Vertical concatenation**
  
  \[ [A;B;C\ldots] \quad \text{vertcat}(A,B,C\ldots) \]

- **Subscript reference**
  
  \[ A(i,j,k\ldots) \quad \text{subsref}(A,S). \text{See help subsref.} \]

- **Subscript assignment**
  
  \[ A(i,j,k\ldots) = B \quad \text{subsasgn}(A,S,B). \text{See help subsasgn.} \]
Note For some toolboxes, the special characters are overloaded, that is, they perform differently in the context of that toolbox. To see the toolboxes that overload a given character, type help followed by the character name. For example, type help transpose. The toolboxes that overload transpose ( . ') are listed. For information about using the character in that toolbox, see the documentation for the toolbox.

See Also

Arithmetic operators +, *, /, \, ^, '

Relational operators <, <=, >, >=, ==, ~=

Elementwise Logical Operators, &, |, Short-Circuit Logical Operators, &&, ||,
**Purpose**
Create vectors, array subscripting, and for loop iterations

**Description**
The colon is one of the most useful operators in MATLAB. It can create vectors, subscript arrays, and specify for iterations.

The colon operator uses the following rules to create regularly spaced vectors:

- $j:k$ is the same as $[j, j+1, \ldots, k]$  
- $j:k$ is empty if $j > k$  
- $j:i:k$ is the same as $[j, j+i, j+2i, \ldots, k]$  
- $j:i:k$ is empty if $i > 0$ and $j > k$ or if $i < 0$ and $j < k$

where $i$, $j$, and $k$ are all scalars.

Below are the definitions that govern the use of the colon to pick out selected rows, columns, and elements of vectors, matrices, and higher-dimensional arrays:

- $A(:,j)$ is the $j$th column of $A$  
- $A(i,:)$ is the $i$th row of $A$  
- $A(:,:,k)$ is the $k$th page of three-dimensional array $A$  
- $A(i,j,k,:)$ is a vector in four-dimensional array $A$. The vector includes $A(i,j,k,1), A(i,j,k,2), A(i,j,k,3)$, and so on.  
- $A(:)$ is all the elements of $A$, regarded as a single column. On the left side of an assignment statement, $A(:)$ fills $A$, preserving its shape from before. In this case, the right side must contain the same number of elements as $A$. 

2-21
Examples

Using the colon with integers,

\[ D = 1:4 \]

results in

\[ D = \\
1 \quad 2 \quad 3 \quad 4 \]

Using two colons to create a vector with arbitrary real increments between the elements,

\[ E = 0:.1:.5 \]

results in

\[ E = \\
0 \quad 0.1000 \quad 0.2000 \quad 0.3000 \quad 0.4000 \quad 0.5000 \]

The command

\[ A(:,:,2) = \text{pascal}(3) \]

generates a three-dimensional array whose first page is all zeros.

\[ A(:,:,1) = \\
0 \quad 0 \quad 0 \\
0 \quad 0 \quad 0 \\
0 \quad 0 \quad 0 \\
\]

\[ A(:,:,2) = \\
1 \quad 1 \quad 1 \\
1 \quad 2 \quad 3 \\
1 \quad 3 \quad 6 \\
\]

See Also

for, linspace, logspace, reshape
Purpose
Absolute value and complex magnitude

Syntax
Y = abs(X)

Description
abs(X) returns an array Y such that each element of Y is the absolute value of
the corresponding element of X.

If X is complex, abs(X) returns the complex modulus (magnitude), which is the
same as

\[ \sqrt{\text{real}(X)^2 + \text{imag}(X)^2} \]

Examples
abs(-5)
ans =
5

abs(3+4i)
ans =
5

See Also
angle, sign, unwrap
Purpose
Construct an array with accumulation

Syntax
A = accumarray(ind, val)
A = accumarray(ind, val, sz)
A = accumarray(ind, val, sz, fun)
A = accumarray(ind, val, sz, fun, fillvalue)

Description
A = accumarray(ind, val) creates an array A from the elements of the vector val, using the corresponding rows of ind as subscripts into A. val must have the same length as the number of rows in ind, unless val is a scalar whose value is repeated for all the rows of ind. If ind is a nonempty column vector, then A is a column vector of length max(ind). If ind is a nonempty matrix with k columns, then A is a k-dimensional array of size max(ind,[],1). If ind is zeros(0,k) with k>1, then A is the k-dimensional empty array of size 0-by-0-by-...-by-0. accumarray accumulates by adding together elements of val at repeated subscripts of A. accumarray fills in A at unspecified subscripts with the value 0.

Note val may be full or sparse and A has the same sparsity as val. If val is sparse and ind is a column vector, then A is the same as sparse(ind,1,val). If val is sparse and ind is a matrix with two columns, then A is the same as sparse(ind(:,1),ind(:,2),val).

A = accumarray(ind, val, sz) creates an array of size sz, where sz is a row vector of nonnegative integer values. If ind is a nonempty column vector, then sz must be [n 1] where n=max(ind). If ind is a nonempty matrix with k columns, then sz must be of length k with all(sz>=max(ind,[],1)). If ind is zeros(0,k) with k>1, then sz must be of length k with all(sz>=0). Nonzero sz resizes A to a nonempty all-zero array.

A = accumarray(ind, val, sz, fun) accumulates values at repeated subscripts of A by applying the function fun, which you specify by a function handle. fun must accept a vector and return a scalar. For example, setting fun=@sum produces the default behavior of accumarray when you do not specify fun.
A = accumarray(ind, val, sz, fun, fillvalue) where val is full, fills in the values of A at unspecified indices with the value fillvalue. If ind is empty, but sz resizes A to nonempty, then all the values of A are fillvalue.

Examples

The following command creates a vector, accumulating at the repeated index 2.

\[
A = \text{accumarray}([1; 2; 2; 4; 5], 11:15)
\]

\[
A = \\
11 \\
25 \\
0 \\
14 \\
15
\]

The following commands create a 3-dimensional array, accumulating at repeated subscript (2,3,4).

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

\[
A = \text{accumarray}({\text{ind}}, 11:14)
\]

\[
A(:,:,1) = \\
11 0 0 \\
0 0 0 \\
\]

\[
A(:,:,2) = \\
0 0 0 \\
12 0 0 \\
\]

\[
A(:,:,3) = \\
0 0 0 \\
0 0 0 \\
\]

\[
A(:,:,4) = \\
0 0 0 \\
\]
The following command repeats the scalar \( \text{val} = \pi \) for all the rows in \( \text{ind} \).

\[
A = \text{accumarray}(\text{ind}, \pi)
\]

\[
A(:,:,1) =
\begin{bmatrix}
3.1416 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

\[
A(:,:,2) =
\begin{bmatrix}
0 & 0 & 0 \\
3.1416 & 0 & 0
\end{bmatrix}
\]

\[
A(:,:,3) =
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

\[
A(:,:,4) =
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 6.2832
\end{bmatrix}
\]

Set

\[
\text{ind} = \begin{bmatrix}
1 & 2 \\
3 & 2 \\
5 & 5 \\
5 & 5
\end{bmatrix}
\]
\[
\text{val} = \begin{bmatrix}
10.1 \\
10.2 \\
10.3 \\
10.4
\end{bmatrix}
\]

The following command does the default summation accumulation at the repeated subscript \((5,5)\).

\[
A = \text{accumarray}(\text{ind}, \text{val});
\]

The following command increases the size of \( A \) beyond \( \max(\text{ind}, [], 1) \).

\[
A = \text{accumarray}(\text{ind}, \text{val}, [6 6]);
\]
The following command uses prod instead of sum as the accumulation function:

\[ A = \text{accumarray}([\text{ind}, \text{val}], [6,6], @\text{prod}); \]

The following command uses max as the accumulation function and fills the values at unspecified subscripts with -Inf.

\[ A = \text{accumarray}([\text{ind}, \text{val}], [6,6], @\text{max}, \text{-Inf}); \]

See Also

full, sparse, sum.
acos

**Purpose**
Inverse cosine, result in radians

**Syntax**
\[ Y = \text{acos}(X) \]

**Description**
\[ Y = \text{acos}(X) \]
returns the inverse cosine (arccosine) for each element of \( X \). For real elements of \( X \) in the domain, \( \text{acos}(X) \) is real and in the range. For real elements of \( X \) outside the domain, \( \text{acos}(X) \) is complex.

The acos function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

**Examples**
Graph the inverse cosine function over the domain.

\[
\begin{align*}
  x &= -1:.05:1; \\
  \text{plot}(x,\text{acos}(x)), \text{grid on}
\end{align*}
\]

**Definition**
The inverse cosine can be defined as

\[
\text{acos} \quad \text{uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.}
\]

**See Also**
acosd, acosh, cos
Purpose  
Inverse cosine, result in degrees

Syntax  
\[ Y = \text{acosd}(X) \]

Description  
\[ Y = \text{acosd}(X) \] is the inverse cosine, expressed in degrees, of the elements of \( X \).

See Also  
cosd, acos
acosh

Purpose
Inverse hyperbolic cosine

Syntax
Y = acosh(X)

Description
Y = acosh(X) returns the inverse hyperbolic cosine for each element of X.
The acosh function operates element-wise on arrays. The function’s domains
and ranges include complex values. All angles are in radians.

Examples
Graph the inverse hyperbolic cosine function over the domain.

```matlab
x = 1:pi/40:pi;
plot(x,acosh(x)), grid on
```

Definition
The hyperbolic inverse cosine can be defined as

Algorithm
acosh uses FDLIBM, which was developed at SunSoft, a Sun Microsystems,
Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see

See Also
acos, cosh
Purpose
Inverse cotangent, result in radians

Syntax
Y = acot(X)

Description
Y = acot(X) returns the inverse cotangent (arccotangent) for each element of X.

The acot function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples
Graph the inverse cotangent over the domains and .

```matlab
x1 = -2*pi:pi/30:-0.1;
x2 = 0.1:pi/30:2*pi;
plot(x1,acot(x1),x2,acot(x2)), grid on
```

Definition
The inverse cotangent can be defined as

Algorithm
acot 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
cot, acotd, acoth
**Purpose**
Inverse cotangent, result in degrees

**Syntax**
\[ Y = \text{acotd}(X) \]

**Description**
\[ Y = \text{acosd}(X) \] is the inverse cotangent, expressed in degrees, of the elements of \( X \).

**See Also**
cotd, acot
Purpose
Inverse hyperbolic cotangent

Syntax
Y = acoth(X)

Description
Y = acoth(X) returns the inverse hyperbolic cotangent for each element of X.
The acoth function operates element-wise on arrays. The function’s domains
and ranges include complex values. All angles are in radians.

Examples
Graph the inverse hyperbolic cotangent over the domains and .

x1 = -30:0.1:-1.1;
x2 = 1.1:0.1:30;
plot(x1,acoth(x1),x2,acoth(x2)), grid on

Definition
The hyperbolic inverse cotangent can be defined as

Algorithm
acoth uses FDLIBM, which was developed at SunSoft, a Sun Microsystems,
Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see

See Also
acot, coth
acsc

Purpose
Inverse cosecant, result in radians

Syntax
Y = acsc(X)

Description
Y = acsc(X) returns the inverse cosecant (arccosecant) for each element of X.
The acsc function operates element-wise on arrays. The function's domains
and ranges include complex values. All angles are in radians.

Examples
Graph the inverse cosecant over the domains and .
   x1 = -10:0.01:-1.01;
   x2 = 1.01:0.01:10;
   plot(x1,acsc(x1),x2,acsc(x2)), grid on

Definition
The inverse cosecant can be defined as

Algorithm
acsc uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc.
business, by Kwok C. Ng, and others. For information about FDLIBM, see

See Also
csc, acscd, acsch
Purpose
Inverse cosecant, result in degrees

Syntax
Y = acscd(X)

Description
Y = acscd(X) is the inverse cotangent, expressed in degrees, of the elements of X.

See Also
cscd, acsc
Purpose
Inverse cosecant and inverse hyperbolic cosecant

Syntax
\[ Y = \text{acsch}(X) \]

Description
\[ Y = \text{acsch}(X) \] returns the inverse hyperbolic cosecant for each element of \( X \).

The \text{acsch} function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

Examples
Graph the inverse hyperbolic cosecant over the domains \(-20\) and \(20\).

\[
\begin{align*}
x1 &= -20:0.01:-1; \\
x2 &= 1:0.01:20; \\
\text{plot}(x1,\text{acsch}(x1),x2,\text{acsch}(x2)), \text{grid on}
\end{align*}
\]

Definition
The hyperbolic inverse cosecant can be defined as

Algorithm
\text{acsch} 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
\text{acsc}, \text{csch}
| **Purpose** | Add a frame to an Audio/Video Interleaved (AVI) file |
| **Syntax** | `aviobj = addframe(aviobj,frame)`  
| | `aviobj = addframe(aviobj,frame1,frame2,frame3,...)`  
| | `aviobj = addframe(aviobj,mov)`  
| | `aviobj = addframe(aviobj,h)`  |
| **Description** | `aviobj = addframe(aviobj,frame)` appends the data in `frame` to the AVI file identified by `aviobj`, which was created by a previous call to `avifile`. `frame` can be either an indexed image (m-by-n) or a truecolor image (m-by-n-by-3) of `double` or `uint8` precision. If `frame` is not the first frame added to the AVI file, it must be consistent with the dimensions of the previous frames.  
| | `addframe` returns a handle to the updated AVI file object, `aviobj`. For example, `addframe` updates the `TotalFrames` property of the AVI file object each time it adds a frame to the AVI file.  
| | `aviobj = addframe(aviobj,frame1,frame2,frame3,...)` adds multiple frames to an AVI file.  
| | `aviobj = addframe(aviobj,mov)` appends the frames contained in the MATLAB movie `mov` to the AVI file `aviobj`. MATLAB movies that store frames as indexed images use the colormap in the first frame as the colormap for the AVI file, unless the colormap has been previously set.  
| | `aviobj = addframe(aviobj,h)` captures a frame from the figure or axis handle `h` and appends this frame to the AVI file. `addframe` renders the figure into an offscreen array before appending it to the AVI file. This ensures that the figure is written correctly to the AVI file even if the figure is obscured on the screen by another window or screen saver.  
| **Note** | If an animation uses XOR graphics, you must use `getframe` to capture the graphics into a frame of a MATLAB movie. You can then add the frame to an AVI movie using the `addframe` syntax `aviobj = addframe(aviobj,mov)`. See the example for an illustration. |
| **Example** | This example calls `addframe` to add frames to the AVI file object `aviobj`. |
addframe

```matlab
fig=figure;
set(fig,'DoubleBuffer','on');
set(gca,'xlim',[-80 80],'ylim',[-80 80],...
    'nextplot','replace','Visible','off')

aviobj = avifile('example.avi')

x = -pi:.1:pi;
radius = 0:length(x);
for i=1:length(x)
    h = patch(sin(x)*radius(i),cos(x)*radius(i),...
        [abs(cos(x(i))) 0 0]);
    set(h,'EraseMode','xor');
    frame = getframe(gca);
    aviobj = addframe(aviobj,frame);
end

aviobj = close(aviobj);
```

See Also

avifile, close, movie2avi
addpath

Purpose
Add directories to MATLAB search path

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

Syntax
addpath('directory')
addpath('dir1','dir2','dir3' ...)
addpath('dir1','dir2','dir3' ...'-flag')
addpath dir1 dir2 dir3 ... -flag

Description
addpath('directory') prepends the specified directory to the current MATLAB search path, that is, adds them to the top of the path. Use the full pathname for directory.

addpath('dir1','dir2','dir3' ...') prepends all the specified directories to the path. Use the full pathname for each dir.

addpath('dir1','dir2','dir3' ...'-flag') either prepends or appends the specified directories to the path depending on the value of flag.

<table>
<thead>
<tr>
<th>flag Argument</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or begin</td>
<td>Prepend specified directories</td>
</tr>
<tr>
<td>1 or end</td>
<td>Append specified directories (add to bottom/end)</td>
</tr>
</tbody>
</table>

dir1 dir2 dir3 ... -flag is the unquoted form of the syntax.

Remarks
To recursively add subdirectories of your directory in addition to the directory itself, run

addpath(genpath('directory'))

Use addpath statements in your startup.m file to use the modified path in future sessions. For details, see “Modifying the Path in a startup.m File”.
Examples

For the current path, viewed by typing `path`,

```matlab
MATLABPATH
c:\matlab\toolbox\general
c:\matlab\toolbox\ops
c:\matlab\toolbox\strfun
```

you can add `c:/matlab/mymfiles` to the front of the path by typing

```matlab
addpath('c:/matlab/mymfiles')
```

Verify that the files were added to the path by typing

```matlab
path
```

and MATLAB returns

```matlab
MATLABPATH
c:\matlab\mymfiles
c:\matlab\toolbox\general
c:\matlab\toolbox\ops
c:\matlab\toolbox\strfun
```

You can also use `genpath` in conjunction with `addpath` to add subdirectories to the path from the command line. For example, to add `/control` and its subdirectories to the path, use

```matlab
addpath(genpath('$matlabroot/toolbox/control'))
```

See Also

genpath, path, pathdef, pathsep, pathtool, rehash, restoredefaultpath, rmpath, savepath, startup

“Search Path” in the MATLAB User Guide
Purpose

Modify date number by field

Syntax

\[ R = \text{adddate}(D, N, F) \]

Description

\( R = \text{adddate}(D, Q, F) \) adds quantity \( Q \) to the indicated date field \( F \) of a serial date number \( D \), returning the updated date number \( R \).

The quantity \( Q \) to be added must be a double scalar whole number, and can be either positive or negative. The date field \( F \) must be a 1-by-\( N \) character array equal to one of the following: 'year', 'month', or 'day'.

If the addition to the date field causes the field to roll over, MATLAB adjusts the next more significant fields accordingly. Adding a negative quantity to the indicated date field rolls back the calendar on the indicated field. If the addition causes the field to roll back, MATLAB adjusts the next less significant fields accordingly.

Examples

Adding 20 days to the given date in late December causes the calendar to roll over to January of the next year:

\[ R = \text{adddate}(\text{datenum}('12/24/1984 12:45'), 20, 'day'); \]

\[
\begin{align*}
\text{datestr}(R) \\
\text{ans} &= \\
13\text{-Jan-1999 12:45}
\end{align*}
\]

See Also
date, datenum, datestr, datevec

References


**Purpose**
Airy functions

**Syntax**

- \( W = \text{airy}(Z) \)
- \( W = \text{airy}(k,Z) \)
- \([W,ierr] = \text{airy}(k,Z)\)

**Definition**
The Airy functions form a pair of linearly independent solutions to

\[
\frac{d^2 W}{dZ^2} - Zw = 0
\]

The relationship between the Airy and modified Bessel functions is

\[
\begin{align*}
Ai(Z) & = \left[ \frac{1}{\pi} \sqrt{\frac{Z}{3}} \right] K_{1/3}(\zeta) \\
Bi(Z) & = \sqrt{\frac{Z}{3}} \left[ I_{-1/3}(\zeta) + I_{1/3}(\zeta) \right]
\end{align*}
\]

where

\[
\zeta = \frac{2}{3} Z^{3/2}
\]

**Description**

- \( W = \text{airy}(Z) \) returns the Airy function, \( Ai(Z) \), for each element of the complex array \( Z \).
- \( W = \text{airy}(k,Z) \) returns different results depending on the value of \( k \).

<table>
<thead>
<tr>
<th>( k )</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The same result as ( \text{airy}(Z) )</td>
</tr>
<tr>
<td>1</td>
<td>The derivative, ( Ai'(Z) )</td>
</tr>
<tr>
<td>2</td>
<td>The Airy function of the second kind, ( Bi(Z) )</td>
</tr>
<tr>
<td>3</td>
<td>The derivative, ( Bi'(Z) )</td>
</tr>
</tbody>
</table>
[W, ierr] = airy(k, Z) also returns completion flags in an array the same size as W.

<table>
<thead>
<tr>
<th>ierr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>airy successfully computed the Airy function for this element.</td>
</tr>
<tr>
<td>1</td>
<td>Illegal arguments</td>
</tr>
<tr>
<td>2</td>
<td>Overflow. Returns Inf</td>
</tr>
<tr>
<td>3</td>
<td>Some loss of accuracy in argument reduction</td>
</tr>
<tr>
<td>4</td>
<td>Unacceptable loss of accuracy, Z too large</td>
</tr>
<tr>
<td>5</td>
<td>No convergence. Returns NaN</td>
</tr>
</tbody>
</table>

See Also

besseli, besselj, besselk, bessely

References


Purpose

Set or query the axes alpha limits

Syntax

\[
\begin{align*}
\text{alpha_limits} & = \text{alim} \\
alim([\text{amin} \ \text{amax}]) & \\
alim\text{_mode} & = \text{alim('mode')} \\
alim('\text{alim\_mode}') & \\
alim(\text{axes\_handle},...) & 
\end{align*}
\]

Description

\[
\begin{align*}
\text{alpha\_limits} & = \text{alim} \text{ returns the alpha limits (the axes ALim property) of the current axes.} \\
alim([\text{amin} \ \text{amax}]) & \text{ sets the alpha limits to the specified values. amin is the value of the data mapped to the first alpha value in the alphamap, and amax is the value of the data mapped to the last alpha value in the alphamap. Data values in between are linearly interpolated across the alphamap, while data values outside are clamped to either the first or last alphamap value, whichever is closest.} \\
alim\text{\_mode} & = \text{alim('mode')} \text{ returns the alpha limits mode (the axes ALimMode property) of the current axes.} \\
alim('\text{alim\_mode}') & \text{ sets the alpha limits mode on the current axes. alim\_mode can be} \\
& \quad \text{auto} \quad \text{MATLAB automatically sets the alpha limits based on the alpha data of the objects in the axes.} \\
& \quad \text{manual} \quad \text{MATLAB does not change the alpha limits.} \\
alim(\text{axes\_handle},...) & \text{ operates on the specified axes.}
\end{align*}
\]

See Also

alpha, alphamap, caxis

Axes ALim and ALimMode properties
Patch FaceVertexAlphaData property
Image and surface AlphaData properties
Transparency for related functions
Transparency in 3-D Visualization for examples
Purpose
Test to determine if all elements are nonzero

Syntax
B = all(A)
B = all(A,dim)

Description
B = all(A) tests whether all the elements along various dimensions of an array are nonzero or logical true (1).

If A is a vector, all(A) returns logical true (1) if all the elements are nonzero and returns logical false (0) if one or more elements are zero.

If A is a matrix, all(A) treats the columns of A as vectors, returning a row vector of 1's and 0's.

If A is a multidimensional array, all(A) treats the values along the first nonsingleton dimension as vectors, returning a logical condition for each vector.

B = all(A,dim) tests along the dimension of A specified by scalar dim.

Examples
Given
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
then B = (A < 0.5) returns logical true (1) only where A is less than one half:
0 0 1 1 1 1 0

The all function reduces such a vector of logical conditions to a single condition. In this case, all(B) yields 0.

This makes all particularly useful in if statements:
if all(A < 0.5)
    do something
end
where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

Applying the all function twice to a matrix, as in all(all(A)), always reduces it to a scalar condition.

```matlab
all(all(eye(3)))
an =
    0
```

**See Also**

any, logical operators (elementwise and short-circuit), relational operators, colon

Other functions that collapse an array's dimensions include max, mean, median, min, prod, std, sum, and trapz.
Purpose
Find all children of specified objects

Syntax
child_handles = allchild(handle_list)

Description
child_handles = allchild(handle_list) returns the list of all children (including ones with hidden handles) for each handle. If handle_list is a single element, allchild returns the output in a vector. Otherwise, the output is a cell array.

Examples
Compare the results returned by these two statements.

get(gca,'Children')
allchild(gca)

See Also
findall, findobj
alpha

Purpose
Set transparency properties for objects in current axes

Syntax
alpha(face_alpha)
alpha(alpha_data)
alpha(alpha_data_mapping)
alpha(object_handle,...)

Description
alpha sets one of three transparency properties, depending on what arguments you specify with the call to this function.

FaceAlpha
alpha(face_alpha) sets the FaceAlpha property of all image, patch, and surface objects in the current axes. You can set face_alpha to

- A scalar — Set the FaceAlpha property to the specified value (for images, set the AlphaData property to the specified value).
- 'flat' — Set the FaceAlpha property to flat.
- 'interp' — Set the FaceAlpha property to interp.
- 'texture' — Set the FaceAlpha property to texture.
- 'opaque' — Set the FaceAlpha property to 1.
- 'clear' — Set the FaceAlpha property to 0.

See Specifying a Single Transparency Value for more information.

AlphaData (Surface Objects)
alpha(alpha_data) sets the AlphaData property of all surface objects in the current axes. You can set alpha_data to

- A matrix the same size as CData — Set the AlphaData property to the specified values.
- 'x' — Set the AlphaData property to be the same as XData.
- 'y' — Set the AlphaData property to be the same as YData.
- 'z' — Set the AlphaData property to be the same as ZData.
- 'color' — Set the AlphaData property to be the same as CData.
• 'rand' — Set the AlphaData property to a matrix of random values equal in size to CData.

**AlphaData (Image Objects)**

alpha(alpha_data) sets the AlphaData property of all image objects in the current axes. You can set alpha_data to

- A matrix the same size as CData — Set the AlphaData property to the specified value.
- 'x' — Ignored.
- 'y' — Ignored.
- 'z' — Ignored.
- 'color' — Set the AlphaData property to be the same as CData.
- 'rand' — Set the AlphaData property to a matrix of random values equal in size to CData.

**FaceVertexAlphaData (Patch Objects)**

alpha(alpha_data) sets the FaceVertexAlphaData property of all patch objects in the current axes. You can set alpha_data to

- A matrix the same size as FaceVertexCData — Set the FaceVertexAlphaData property to the specified value.
- 'x' — Set the FaceVertexAlphaData property to be the same as Vertices(:,1).
- 'y' — Set the FaceVertexAlphaData property to be the same as Vertices(:,2).
- 'z' — Set the FaceVertexAlphaData property to be the same as Vertices(:,3).
- 'color' — Set the FaceVertexAlphaData property to be the same as FaceVertexCData.
- 'rand' — Set the FaceVertexAlphaData property to random values.

See Mapping Data to Transparency for more information.
**alpha**

**AlphaDataMapping**

alpha(alpha_data_mapping) sets the AlphaDataMapping property of all image, patch, and surface objects in the current axes. You can set alpha_data_mapping to

- 'scaled' — Set the AlphaDataMapping property to scaled.
- 'direct' — Set the AlphaDataMapping property to direct.
- 'none' — Set the AlphaDataMapping property to none.

alpha(object_handle,value) sets the transparency property only on the object identified by object_handle.

**See Also**

alim, alphamap

Image: AlphaData, AlphaDataMapping

Patch: FaceAlpha, FaceVertexAlphaData, AlphaDataMapping

Surface: FaceAlpha, AlphaData, AlphaDataMapping

Transparency for related functions

Transparency in 3-D Visualization for examples
Purpose

Specify the figure alphamap (transparency)

Syntax

alphamap(alpha_map)
alphamap('parameter')
alphamap('parameter',length)
alphamap('parameter',delta)
alphamap(figure_handle,...)

alpha_map = alphamap
alpha_map = alphamap(figure_handle)
alpha_map = alphamap('parameter')

Description

alphamap enables you to set or modify a figure's AlphaMap property. Unless you specify a figure handle as the first argument, alphamap operates on the current figure.

alphamap(alpha_map) sets the AlphaMap of the current figure to the specified m-by-1 array of alpha values.

alphamap('parameter') creates a new alphamap or modifies the current alphamap. You can specify the following parameters:

- default — Set the AlphaMap property to the figure's default alphamap.
- rampup — Create a linear alphamap with increasing opacity (default length equals the current alphamap length).
- rampdown — Create a linear alphamap with decreasing opacity (default length equals the current alphamap length).
- vup — Create an alphamap that is opaque in the center and becomes more transparent linearly towards the beginning and end (default length equals the current alphamap length).
- vdown — Create an alphamap that is transparent in the center and becomes more opaque linearly towards the beginning and end (default length equals the current alphamap length).
- increase — Modify the alphamap making it more opaque (default delta is .1, which is added to the current values).
- decrease — Modify the alphamap making it more transparent (default delta is .1, which is subtracted from the current values).
• spin — Rotate the current alphamap (default delta is 1; note that delta must be an integer).

alphamap('parameter',length) creates a new alphamap with the length specified by length (used with parameters rampup, rampdown, vup, vdown).

alphamap('parameter',delta) modifies the existing alphamap using the value specified by delta (used with parameters increase, decrease, spin).

alphamap(figure_handle,...) performs the operation on the alphamap of the figure identified by figure_handle.

alpha_map = alphamap returns the current alphamap.

alpha_map = alphamap(figure_handle) returns the current alphamap from the figure identified by figure_handle.

alpha_map = alphamap('parameter') returns the alphamap modified by the parameter, but does not set the AlphaMap property.

See Also

alim, alpha

Image: AlphaData, AlphaDataMapping
Patch: FaceAlpha, AlphaData, AlphaDataMapping
Surface: FaceAlpha, AlphaData, AlphaDataMapping

Transparency for related functions

Transparency in 3-D Visualization for examples

area(Y)
grid on
colormap summer
set(gca,'Layer','top')
**Purpose**
Get ancestor of graphics object

**Syntax**

\[
p = \text{ancestor}(h, \text{type})
\]

\[
p = \text{ancestor}(h, \text{type}, 'toplevel')
\]

**Description**

\[
p = \text{ancestor}(h, \text{type})\]
returns the handle of the closest ancestor of \(h\), if the ancestor is one of the types of graphics objects specified by type. type can be:

- a string that is the name of a single type of object. For example, ‘figure’
- a cell array containing the names of multiple objects. For example, \{'hgtransform','hggroup','axes'}\)

If MATLAB cannot find an ancestor of \(h\) that is one of the specified types, then ancestor returns \(p\) as empty.

Note that ancestor returns \(p\) as empty but does not issue an error if \(h\) is not the handle of a Handle Graphics object.

\[
p = \text{ancestor}(h, \text{type}, 'toplevel')\]
returns the highest-level ancestor of \(h\), if this type appears in the type argument.

**Examples**

Create some line objects and parent them to an hggroup object.

```matlab
hgg = hggroup;
hgl = line(randn(5),randn(5),'Parent',hgg);
```

Now get the ancestor of the lines.

```matlab
p = ancestor(hgg,{'figure','axes','hggroup'});
get(p,'Type')
an =

hggroup
```

Now get the top-level ancestor

```matlab
p = ancestor(hgg,{'figure','axes','hggroup'},'toplevel');
get(p,'type')
an =

figure
ancestor

**See Also**

findobj
Purpose

Phase angle

Syntax

\[ P = \text{angle}(Z) \]

Description

\( P = \text{angle}(Z) \) returns the phase angles, in radians, for each element of complex array \( Z \). The angles lie between \( \pm \pi \).

For complex \( Z \), the magnitude \( R \) and phase angle \( \theta \) are given by

\[
R = \text{abs}(Z) \\
\theta = \text{angle}(Z)
\]

and the statement

\[ Z = R.*\text{exp}(i*\theta) \]

converts back to the original complex \( Z \).

Examples

\[
Z = [\begin{array}{cccc}
1 - 1i & 2 + 1i & 3 - 1i & 4 + 1i \\
1 + 2i & 2 - 2i & 3 + 2i & 4 - 2i \\
1 - 3i & 2 + 3i & 3 - 3i & 4 + 3i \\
1 + 4i & 2 - 4i & 3 + 4i & 4 - 4i
\end{array}] \\
\]

\[ P = \text{angle}(Z) \]

\[
P = \\
-0.7854 & 0.4636 & -0.3218 & 0.2450 \\
1.1071 & -0.7854 & 0.5880 & -0.4636 \\
-1.2490 & 0.9828 & -0.7854 & 0.6435 \\
1.3258 & -1.1071 & 0.9273 & -0.7854
\]

Algorithm

The \( \text{angle} \) function can be expressed as

\( \text{angle}(z) = \text{imag}(\log(z)) = \text{atan2}(\text{imag}(z), \text{real}(z)) \).

See Also

abs, \text{atan2}, \text{unwrap}
**Purpose**
Create annotation objects

**Syntax**

```
annotation(annotation_type)
annotation('line',x,y)
annotation('arrow',x,y)
annotation('doublearrow',x,y)
annotation('textarrow',x,y)
annotation('textbox',[x y w h])
annotation('ellipse',[x y w h])
annotation('rectangle',[x y w h])
annotation(figure_handle,...)
annotation(...,'PropertyName',PropertyValue,...)
anno_obj_handle = annotation(...)
```

**Description**

`annotation(annotation_type)` creates the specified annotation type using default values for all properties. `annotation_type` can be one of the following strings:

- `line`, `arrow`, `doublearrow` (two-headed arrow), `textarrow` (arrow with attached textbox), `textbox`, `ellipse`, or `rectangle`.

`annotation('line',x,y)` creates a line annotation object that extends from the point defined by \(x(1),y(1)\) to the point defined by \(x(2),y(2)\), specified in normalized figure units.

`annotation('arrow',x,y)` creates an arrow annotation object that extends from the point defined by \(x(1),y(1)\) to the point defined by \(x(2),y(2)\), specified in normalized figure units.

`annotation('doublearrow',x,y)` creates a two-headed annotation object that extends from the point defined by \(x(1),y(1)\) to the point defined by \(x(2),y(2)\), specified in normalized figure units.

`annotation('textarrow',x,y)` creates a textarrow annotation object that extends from the point defined by \(x(1),y(1)\) to the point defined by \(x(2),y(2)\), specified in normalized figure units. The tail end of the arrow is attached to an editable textbox.
annotation('textbox',[x y w h]) creates an editable textbox annotation with its lower-left corner at the point \(x, y\), a width \(w\), and a height \(h\), specified in normalized figure units. Specify \(x\), \(y\), \(w\), and \(h\) in a single vector.

To type into the textbox, enable plot edit mode (plotedit) and double click within the box.

annotation('ellipse',[x y w h]) creates an ellipse annotation with the lower-left corner of the bounding rectangle at the point \(x, y\), a width \(w\), and a height \(h\), specified in normalized figure units. Specify \(x\), \(y\), \(w\), and \(h\) in a single vector.

annotation('rectangle',[x y w h]) creates a rectangle annotation with the lower-left corner of the rectangle at the point \(x, y\), a width \(w\), and a height \(h\), specified in normalized figure units. Specify \(x\), \(y\), \(w\), and \(h\) in a single vector.

annotation(figure_handle,...) creates the annotation in the specified figure.

annotation(...,'PropertyName',PropertyValue,...) creates the annotation and sets the specified properties to the specified values.

anno_obj_handle = annotation(...) returns the handle to the annotation object that is created.

Annotation Layer

All annotation objects are displayed in an overlay axes that covers the figure. This layer is designed to display only annotation objects. You should not parent objects to this axes or set any properties of this axes. See the See Also section for information on the properties of annotation objects that you can set.

Objects in the Plotting Axes
You can create lines, text, rectangles, and ellipses in data coordinates in the axes of a graph using the `line`, `text`, and `rectangle` functions. These objects are not placed in the annotation axes and must be located inside their parent axes.

Normalized Coordinates
Annotation objects use normalize coordinates to specify locations within the figure. In normalized coordinates, the point 0,0 is always the lower-left corner
and the point 1,1 is always the upper-right corner of the figure window regardless of the figure size.

**See Also**

Properties for the annotation objects: arrow, doublearrow, ellipse, line, rectangle, textarrow, textbox

See Annotating Graphs and Annotation Objects for more information.
You can set and query annotation object properties using the set and get functions and the Property Editor (displayed with the `propertyeditor` command).

Use the annotation function to create annotation objects and obtain their handles.

Properties You Can Modify
This section lists the properties you can modify on an annotation ellipse object.

- **Color**  
  ColorSpec Default: [0 0 0]  
  Color of the arrow. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.  
  See the ColorSpec reference page for more information on specifying color.

- **HeadLength**  
  scalar value in points  
  Length of the arrow head. Specify this property in points (1 point = 1/72 inch).  
  See also `HeadWidth`.

- **HeadStyle**  
  select string from list  
  Style of the arrow head. Specify this property as one of the strings from the following table.

<table>
<thead>
<tr>
<th>Head Style String</th>
<th>Head</th>
<th>Head Style String</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>![none]</td>
<td>star4</td>
<td>![star4]</td>
</tr>
<tr>
<td>plain</td>
<td>![plain]</td>
<td>rectangle</td>
<td>![rectangle]</td>
</tr>
<tr>
<td>ellipse</td>
<td>![ellipse]</td>
<td>diamond</td>
<td>![diamond]</td>
</tr>
<tr>
<td>vback1</td>
<td>![vback1]</td>
<td>rose</td>
<td>![rose]</td>
</tr>
<tr>
<td>vback2 (Default)</td>
<td>![vback2]</td>
<td>hypocycloid</td>
<td>![hypocycloid]</td>
</tr>
</tbody>
</table>
Annotation Arrow Properties

<table>
<thead>
<tr>
<th>Head Style String</th>
<th>Head</th>
<th>Head Style String</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>vback3</td>
<td>➩</td>
<td>astroid</td>
<td>➩</td>
</tr>
<tr>
<td>cbback1</td>
<td>➩</td>
<td>deltoid</td>
<td>➩</td>
</tr>
<tr>
<td>cbback2</td>
<td>➩</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbback3</td>
<td>➩</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**HeadWidth**  
Scalar value in points

*Width of the arrow head.* Specify this property in points (1 point = 1/72 inch). See also **HeadLength**.

**LineStyle**  
{-} | -- | : | -. | none

*Line style.* This property specifies the line style of the arrow stem. Available line styles are shown in the following table.

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

**LineWidth**  
Scalar

*The width of the arrow stem.* Specify this value in points (1 point = 1/72 inch). The default **LineWidth** is 0.5 points.

**X**  
Vector [Xbegin, Xend]

*X-coordinates of the beginning and ending points for arrow.* Specify this property as a vector of x-axis (horizontal) values that specify the beginning and ending points of the arrow, units normalized to the figure.
**Annotation Arrow Properties**

\[ Y \text{ vector } [Y_{\text{begin}} \ Y_{\text{end}}] \]

*Y-coordinates of the beginning and ending points for arrow.* Specify this property as a vector of y-axis (vertical) values that specify the beginning and ending points of the arrow, units normalized to the figure.
Annotation Doublearrow Properties

Modifying Properties
You can set and query annotation object properties using the set and get functions and the Property Editor (displayed with the propertyeditor command).

Use the annotation function to create annotation objects and obtain their handles.

Annotation Doublearrow Property Descriptions

Properties You Can Modify
This section lists the properties you can modify on an annotation doublearrow object.

Color
ColorSpec Default: [0 0 0]
Color of the doublearrow. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the ColorSpec reference page for more information on specifying color.

Head1Length
Scalar value in points
Length of the first arrow head. Specify this property in points (1 point = 1/72 inch). See also Head1Width.

The first arrow head is located at the end defined by the point x(1), y(1). See also the X and Y properties.

Head2Length
Scalar value in points
Length of the second arrow head. Specify this property in points (1 point = 1/72 inch). See also Head1Width.

The first arrow head is located at the end defined by the point x(end), y(end). See also the X and Y properties.

Head1Style
Select string from list
Style of the first arrow head. Specify this property as one of the strings from the following table

Head2Style
Select string from list
Style of the second arrow head. Specify this property as one of the strings from the following table.
Annotation Doublearrow Properties

<table>
<thead>
<tr>
<th>Head Style String</th>
<th>Head</th>
<th>Head Style String</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>star4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>plain</td>
<td>rectangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ellipse</td>
<td>diamond</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vback1</td>
<td>rose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vback2 (Default)</td>
<td>hypocycloid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vback3</td>
<td>astroid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbback1</td>
<td>deltoid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbback2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbback3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Head1Width**

Scalar value in points

*Width of the first arrow head.* Specify this property in points (1 point = 1/72 inch). See also Head1Length.

**Head2Width**

Scalar value in points

*Width of the second arrow head.* Specify this property in points (1 point = 1/72 inch). See also Head2Length.
**Annotation Doublearrow Properties**

**LineStyle**

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

*Line style.* This property specifies the line style of the doublearrow stem. Available line styles are shown in the following table.

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

**LineWidth**

scalar

*The width of the arrow stem.* Specify this value in points (1 point = \(\frac{1}{72}\) inch). The default LineWidth is 0.5 points.

**X**

vector \([X_{\text{begin}} \ X_{\text{end}}]\)

*X-coordinates of the beginning and ending points for doublearrow.* Specify this property as a vector of x-axis (horizontal) values that specify the beginning and ending points of the doublearrow, units normalized to the figure.

**Y**

vector \([Y_{\text{begin}} \ Y_{\text{end}}]\)

*Y-coordinates of the beginning and ending points for doublearrow.* Specify this property as a vector of y-axis (vertical) values that specify the beginning and ending points of the doublearrow, units normalized to the figure.
You can set and query annotation object properties using the set and get functions and the Property Editor (displayed with the propertyeditor command).

Use the annotation function to create annotation objects and obtain their handles.

### Properties You Can Modify

This section lists the properties you can modify on an annotation ellipse object.

- **EdgeColor**
  
  ColorSpec Default: [0 0 0]
  
  *Color of the ellipse edge.* A three-element RGB vector or one of the MATLAB predefined names, specifying the edge color.

  See the ColorSpec reference page for more information on specifying color.

- **FaceColor**
  
  ColorSpec Default: [0 0 0]
  
  *Color of the ellipse interior.* A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the interior of the ellipse.

  See the ColorSpec reference page for more information on specifying color.

- **Height**
  
  *Vertical dimension in normalized units.*

  *Vertical dimension of the ellipse.* This property specifies height of the ellipse in units normalized to the figure.

- **LineStyle**
  
  \{-\} | -- | : | --. | none

  *Line style.* This property specifies the line style of the ellipse edge. Available line styles are shown in the following table.

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

**LineWidth**

 scalar

 *The width of the ellipse edge.* Specify this value in points (1 point = \( \frac{1}{72} \) inch). The default `LineWidth` is 0.5 points.

**Width**

Horizontal dimension in normalized units

*Horizontal dimension of the ellipse.* This property specifies width of the ellipse in units normalized to the figure.

Note that, if `Width` and `Height` are equal, the ellipse becomes a circle when the figure width and height (last two elements in the figure `Position` property vector) are also equal.

**X**

Horizontal dimension in normalized units

*Horizontal dimension of the ellipse.* This property specifies the horizontal location of the center of the ellipse, in units normalized to the figure.

**Y**

Vertical dimension in normalized units

*Vertical dimension of the ellipse.* This property specifies the vertical location of the center of the ellipse, in units normalized to the figure.
Annotation Line Properties

Modifying Properties

You can set and query annotation object properties using the set and get functions and the Property Editor (displayed with the propertyeditor command).

Use the annotation function to create annotation objects and obtain their handles.

Properties You Can Modify

This section lists the properties you can modify on an annotation ellipse object.

Color

ColorSpec Default: [0 0 0]

Color of the line. 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.

LineStyle

{−}  |  −−  |  :  |  −  |  none

Line style. This property specifies the line style. Available line styles are shown in the following table.

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

LineWidth

scalar

The width of the line. Specify this value in points (1 point = 1/72 inch). The default LineWidth is 0.5 points.

X

vector [Xbegin Xend]

X-coordinates of the beginning and ending points for line. Specify this property as a vector of x-axis (horizontal) values that specify the beginning and ending points of the line, units normalized to the figure.
**Annotation Line Properties**

\( \mathbf{Y} \) \text{ vector } [Y_{\text{begin}}, Y_{\text{end}}]

*Y-coordinates of the beginning and ending points for arrow.* Specify this property as a vector of y-axis (vertical) values that specify the beginning and ending points of the line, units normalized to the figure.
You can set and query annotation object properties using the set and get functions and the Property Editor (displayed with the propertyeditor command).

Use the annotation function to create annotation objects and obtain their handles.

Properties You Can Modify
This section lists the properties you can modify on an annotation ellipse object.

**EdgeColor**
ColorSpec Default: [0 0 0]
Color of the rectangle edge. A three-element RGB vector or one of the MATLAB predefined names, specifying the edge color.

See the ColorSpec reference page for more information on specifying color.

**FaceColor**
ColorSpec Default: [0 0 0]
Color of the rectangle interior. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the interior of the rectangle.

See the ColorSpec reference page for more information on specifying color.

**Height**
Vertical dimension in normalized units
Vertical dimension of the rectangle. This property specifies height of the rectangle in units normalized to the figure.

**LineStyle**
{-} | −− | : | − . | none
Line style. This property specifies the line style of the rectangle edge. Available line styles are shown in the following table.

<table>
<thead>
<tr>
<th>Specifier String</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>Solid line (default)</td>
</tr>
<tr>
<td>−−</td>
<td>Dashed line</td>
</tr>
<tr>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>− .</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>
**Annotation Rectangle Properties**

**LineWidth** scalar
The width of the rectangle edge. Specify this value in points (1 point = \( \frac{1}{72} \) inch). The default LineWidth is 0.5 points.

**Width** horizontal dimension in normalized units
*Horizontal dimension of the ellipse.* This property specifies width of the ellipse in units normalized to the figure.

Note that, if Width and Height are equal, the ellipse becomes a circle when the figure width and height (last two elements in the figure Position property vector) are also equal.

**X** horizontal dimension in normalized units
*Horizontal dimension of the ellipse.* This property specifies the horizontal location of the center of the ellipse, in units normalized to the figure.

**Y** vertical dimension in normalized units
*Horizontal dimension of the ellipse.* This property specifies the vertical location of the center of the ellipse, in units normalized to the figure.
You can set and query annotation object properties using the set and get functions and the Property Editor (displayed with the propertyeditor command).

Use the annotation function to create annotation objects and obtain their handles.

Properties You Can Modify
This section lists the properties you can modify on an annotation ellipse object.

**Color**
ColorSpec Default: [0 0 0]
Color of the arrow, text and text border. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the arrow, the color of the text (TextColor property), and the rectangle enclosing the text (TextEdgeColor property).

Setting the Color property also sets the TextColor and TextEdgeColor properties to the same color. However, if the value of the TextEdgeColor is none, it remains none and the text box is not displayed. You can set TextColor or TextEdgeColor independently without affecting other properties.

For example, if you want to create a textarrow with a red arrow and black text in a black box, you must:

1. Set the Color property to red — `set(h, 'Color', 'r')`
2. Set the TextColor to black — `set(h, 'TextColor', 'k')`
3. Set the TextEdgeColor to black. — `set(h, 'TextEdgeColor', 'k')`

If you do not want display the text box, set the TextEdgeColor to none.

See the ColorSpec reference page for more information on specifying color.

**FontName**
A name, such as Helvetica
Font family. A string specifying the name of the font to use for the text. To display and print properly, this font must be supported on your system. The default font is Helvetica.

**FontSize**
size in points
Approximate size of text characters. A value specifying the font size to use in points. The default size is 10 (1 point = 1/72 inch).
Annotation Textarrow Properties

**FontWeight** light | {normal} | demi | bold

*Weight of text characters.* MATLAB uses this property to select a font from those available on your system. Generally, setting this property to *bold* or *demi* causes MATLAB to use a bold font.

**HeadLength** scalar value in points

*Length of the arrow head.* Specify this property in points (1 point = 1/72 inch). See also **HeadWidth**.

**HeadStyle** select string from list

*Style of the arrow head.* Specify this property as one of the strings from the following table.

<table>
<thead>
<tr>
<th>Head Style String</th>
<th>Head</th>
<th>Head Style String</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td>star4</td>
<td></td>
</tr>
<tr>
<td>plain</td>
<td></td>
<td>rectangle</td>
<td></td>
</tr>
<tr>
<td>ellipse</td>
<td></td>
<td>diamond</td>
<td></td>
</tr>
<tr>
<td>vback1</td>
<td></td>
<td>rose</td>
<td></td>
</tr>
<tr>
<td>vback2 (Default)</td>
<td></td>
<td>hypocycloid</td>
<td></td>
</tr>
<tr>
<td>vback3</td>
<td></td>
<td>astroid</td>
<td></td>
</tr>
<tr>
<td>cback1</td>
<td></td>
<td>deltoid</td>
<td></td>
</tr>
<tr>
<td>cback2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cback3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Annotation Textarrow Properties

**HeadWidth**
scalar value in points

*Width of the arrow head.* Specify this property in points (1 point = 1/72 inch). See also HeadLength.

**HorizontalAlignment**
{left} | center | right

*Horizontal alignment of text.* This property specifies the horizontal alignment of the text with respect to the arrow.

**Interpreter**
{tex} | latex | none

*Interpret $\TeX$ instructions.* This property controls whether MATLAB interprets certain characters in the **String** property as $\TeX$ instructions (default) or displays all characters literally. See the text object **String** property for a list of supported $\TeX$ instructions.

To enable a complete $\TeX$ interpreter for text objects, set the **Interpreter** property to **latex**.

**LineStyle**
{-} | -- | : | -. | none

*Line style.* This property specifies the line style of the arrow stem. Available line styles are shown in the following table.

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

**LineWidth**
scalar

*The width of the arrow stem.* Specify this value in points (1 point = 1/72 inch). The default **LineWidth** is 0.5 points.

**String**
string

*The text string.* Specify this property as a quoted string for single-line strings, or as a cell array of strings for multiline strings. MATLAB displays this string
in the text box with the specified HorizontalAlignment and VerticalAlignment. See the Interpreter property for information on using TeX characters.

**TextBackgroundColor**

ColorSpec Default: none

*Color of text background rectangle.* A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the ColorSpec reference page for more information on specifying color.

**TextColor**

ColorSpec Default: [0 0 0]

*Color of text.* A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the ColorSpec reference page for more information on specifying color. Setting the Color property also sets this property.

**TextEdgeColor**

ColorSpec or none Default: none

*Color of edge of text rectangle.* A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the rectangle that encloses the text.

See the ColorSpec reference page for more information on specifying color. Setting the Color property also sets this property.

**TextLineWidth**

width in points

*The width of the text rectangle edge.* Specify this value in points (1 point = 1/72 inch). The default LineWidth is 0.5 points.

**TextMargin**

dimension in pixels default: 5

*Space around text.* Specify a value in pixels that defines the space around the text string, but within the TextEdgeColor rectangle.

**TextRotation**

rotation angle in degrees (default = 0)

*Text orientation.* This property determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation). Angles do not accumulate; a rotation of 0 degrees is always horizontal.

**VerticalAlignment**

top | cap | {middle} | baseline | bottom

*Vertical alignment of text.* This property specifies the vertical alignment of the text with respect to the arrow. The possible values mean
Annotation Textarrow Properties

- **top** — Place the top of the string at the specified y-position.
- **cap** — Place the string so that the top of a capital letter is at the y-position.
- **middle** — Place the middle of the string at the y-position.
- **baseline** — Place font baseline at the y-position.
- **bottom** — Place the bottom of the string at the y-position.

\[ X \quad \text{vector} \ [X_{\text{begin}} \ X_{\text{end}}] \]

*Beginning and ending points for arrow.* Specify this property as a vector of x-axis (horizontal) values that specify the beginning and ending points of the arrow, units normalized to the figure.

\[ Y \quad \text{vector} \ [Y_{\text{begin}} \ Y_{\text{end}}] \]

*Beginning and ending points for arrow.* Specify this property as a vector of y-axis (vertical) values that specify the beginning and ending points of the arrow, units normalized to the figure.
You can set and query annotation object properties using the \texttt{set} and \texttt{get} functions and the Property Editor (displayed with the \texttt{propertyeditor} command).

Use the annotation function to create annotation objects and obtain their handles.

This section lists the properties you can modify on an annotation ellipse object.

\textbf{BackgroundColor} \hspace{1em} \texttt{ColorSpec Default: [0 0 0]}

\textit{Color of textbox background}. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color of the textbox. A value of \texttt{none} makes the textbox transparent, enabling objects behind the textbox to be visible.

\textbf{Color} \hspace{1em} \texttt{ColorSpec Default: [0 0 0]}

\textit{Color of the text}. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the \texttt{ColorSpec} reference page for more information on specifying color.

\textbf{EdgeColor} \hspace{1em} \texttt{ColorSpec Default: [0 0 0]}

\textit{Color of the textbox edge}. A three-element RGB vector or one of the MATLAB predefined names, specifying the edge color.

See the \texttt{ColorSpec} reference page for more information on specifying color.

\textbf{FaceAlpha} \hspace{1em} \texttt{Scalar alpha value in range [0 1]}

\textit{Transparency of textbox background}. This property defines the degree to which the textbox background color is transparent. A value of 1 (the default) makes the color opaque, a value of 0 makes the background completely transparent (i.e., invisible). The default FaceAlpha is 1.

\textbf{FitHeightToText} \hspace{1em} \texttt{on | {off}}

\textit{Automatically adjust textbox height to fit text}. MATLAB automatically wraps text strings to fit the width of the textbox. However, if the text string is long enough, it extends beyond the bottom of the textbox.
When you set this mode to on, MATLAB automatically adjusts the height of the textbox to accommodate the string.

The fit-height-to-text behavior continues to apply if you resize the textbox from the two side handles.
Annotation Textbox Properties

However, if you resize the textbox from any other handles, the position you set is honored without regard to how the text fits the box.

FontAngle

*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

*Font family.* A string specifying the name of the font to use for the textbox object. To display and print properly, this font must be supported on your system. The default font is *Helvetica.*

FontSize

*Approximate size of text characters.* A value specifying the font size to use in points. The default size is 10 (1 point = 1/72 inch).

FontWeight

*Weight of text characters.* MATLAB uses this property to select a font from those available on your system. Generally, setting this property to *bold* or *demi* causes MATLAB to use a bold font.

HorizontalAlignment

*Horizontal alignment of text.* This property specifies the horizontal justification of the textbox string. It determines where MATLAB places the string with respect to the value of the Position property’s x value (the first element in the position vector).
Annotation Textbox Properties

**Interpreter**  
{tex} | latex | none

Interpret TeX instructions. This property controls whether MATLAB interprets certain characters in the String property as TeX instructions (default) or displays all characters literally. See the text object String property for a list of supported TeX instructions.

To enable a complete TeX interpreter for text objects, set the Interpreter property to latex.

**LineStyle**  
{−} | −− | : | −. | none

Line style of edge. This property specifies the line style of the textbox edge. Available line styles are shown in the following table.

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

**LineWidth**  
scalar

The width of the textbox edge. Specify this value in points (1 point = $\frac{1}{72}$ inch). The default LineWidth is 0.5 points.

**Margin**  
scalar pixel value

Space around text. Specify a value in pixels that defines the space around the text string, but within the textbox.

**Position**  
four-element vector [x, y, width, height]

Size and location of textbox. Specify the lower-left corner of the textbox with the first two elements of the vector defining the point x, y. The third and fourth elements specify the width and height respectively.
Annotation Textbox Properties

String

String. Specify this property as a quoted string for single-line strings, or as a cell array of strings for multiline strings. MATLAB displays this string at the specified Position. See the Interpreter property for more information on using \TeX\ characters.

VerticalAlignment top | cap | {middle} | baseline | bottom

Vertical alignment of text within textbox. This property specifies the vertical alignment of the text in the textbox. It determines where MATLAB places the string with respect to the value of the Position property’s y value (the second element in the position vector). The possible values mean

• top — Place the top of the string at the specified y-position.
• cap — Place the string so that the top of a capital letter is at the y-position.
• middle — Place the middle of the string at the y-position.
• baseline — Place font baseline at the y-position.
• bottom — Place the bottom of the string at the y-position.
Purpose  The most recent answer
Syntax   ans
Description  MATLAB creates the ans variable automatically when you specify no output argument.
Examples  The statement
          2+2
          is the same as
          ans = 2+2
See Also  display
any

**Purpose**
Test for any nonzeros

**Syntax**

\[
B = \text{any}(A)
\]

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

**Description**

\( B = \text{any}(A) \) tests whether any of the elements along various dimensions of an array are nonzero or logical true (1). If \( A \) is a vector, \( \text{any}(A) \) returns logical true (1) if any of the elements of \( A \) are nonzero, and returns logical false (0) if all the elements are zero.

If \( A \) is a matrix, \( \text{any}(A) \) treats the columns of \( A \) as vectors, returning a row vector of 1’s and 0’s.

If \( A \) is a multidimensional array, \( \text{any}(A) \) treats the values along the first nonsingleton dimension as vectors, returning a logical condition for each vector.

\( B = \text{any}(A, \text{dim}) \) tests along the dimension of \( A \) specified by scalar \( \text{dim} \).

<table>
<thead>
<tr>
<th>1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
</tr>
</tbody>
</table>

\[ A \]

\[ \text{any}(A, 1) \]

\[ \text{any}(A, 2) \]

**Examples**

Given

\[ A = \begin{bmatrix} 0.53 & 0.67 & 0.01 & 0.38 & 0.07 & 0.42 & 0.69 \end{bmatrix} \]

then \( B = (A < 0.5) \) returns logical true (1) only where \( A \) is less than one half:

\[ 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \]

The any function reduces such a vector of logical conditions to a single condition. In this case, \( \text{any}(B) \) yields 1.

This makes any particularly useful in if statements:

```matlab
if \text{any}(A < 0.5)
    \text{do something}
end
```
where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

Applying the any function twice to a matrix, as in \( \text{any(any}(A)) \), always reduces it to a scalar condition.

```matlab
any(any(eye(3)))
ans =
    1
```

**See Also**

all, logical operators (elementwise and short-circuit), relational operators, colon

Other functions that collapse an array's dimensions include max, mean, median, min, prod, std, sum, and trapz.
area

Purpose
Filled area 2-D plot

Syntax
area(Y)
area(X,Y)
area(...,basevalue)
area(...,'PropertyName',PropertyValue,...)
area(axes_handle,...)
h = area(...)
area('v6',...)

Description
An area graph displays elements in Y as one or more curves and fills the area beneath each curve. When Y is a matrix, the curves are stacked showing the relative contribution of each row element to the total height of the curve at each x interval.

area(Y) plots the vector Y or the sum of each column in matrix Y. The x-axis automatically scales to 1:size(Y,1).

area(X,Y) For vectors X and Y, area(X,Y) is the same as plot(X,Y) except that the area between 0 and Y is filled. When Y is a matrix, area(X,Y) plots the columns of Y as filled areas. For each X, the net result is the sum of corresponding values from the columns of Y.

If X is a vector, length(X) must equal length(Y) and X must be monotonic. If X is a matrix, size(X) must equal size(Y) and each column of X must be monotonic. To make a vector or matrix monotonic, use sort.

area(...,basevalue) specifies the base value for the area fill. The default basevalue is 0. See the BaseValue property for more information.

area(...,'PropertyName',PropertyValue,...) specifies property name and property value pairs for the patch graphics object created by area.

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

h = area(...) returns handles of areaseries graphics objects.
Backward Compatible Version

`hpatches = area('v6',...)` returns the handles of patch objects instead of `areaseries` objects for compatibility with MATLAB 6.5 and earlier. See patch object properties for a discussion of the properties you can set to control the appearance of these area graphs.

See Plot Objects and Backward Compatibility for more information.

Areaseries Objects

Creating an area graph of an \( m \)-by-\( n \) matrix creates \( n \) `areaseries` objects (i.e., one per column), whereas a 1-by-\( n \) vector creates one area object.

Note that some `areaseries` object properties that you set on an individual `areaseries` object set the value for all `areaseries` objects in the graph. See the property descriptions for information on specific properties.

Examples Stacked Area Graph

This example plots the data in the variable \( Y \) as an area graph. Each subsequent column of \( Y \) is stacked on top of the previous data. Note that the figure colormap controls the coloring of the individual areas. You can explicitly set the color of an area using the `EdgeColor` and `FaceColor` properties.

\[
Y = [1, 5, 3; 3, 2, 7; 1, 5, 3; 2, 6, 1];
\]

`area(Y)`

`grid on`  
`colormap summer`  
`set(gca,'Layer','top')`  
`title 'Stacked Area Plot'`
Adjusting the Base Value

The area function uses a y-axis value of 0 as the base of the filled areas. You can change this value by setting the area `BaseValue` property. For example, negate one of the values of `Y` from the previous example and replot the data.

```matlab
Y(3,1) = -1;  \% Was 1
h = area(Y);
set(gca,'Layer','top')
grid on
colormap summer
```

The area graph now looks like this:
Adjusting the `BaseValue` property improves the appearance of the graph:

```matlab
set(h, 'BaseValue', -2)
```

Note that setting the `BaseValue` property on one `areaseries` object sets the values of all objects.
Specifying Colors and Line Styles
You can specify the colors of the filled areas and the type of lines used to separate them.

```matlab
h = area(Y, -2); % Set BaseValue via argument
set(h(1), 'FaceColor', [.5 0 0])
set(h(2), 'FaceColor', [.7 0 0])
set(h(3), 'FaceColor', [1 0 0])
set(h, 'LineStyle', ':', 'LineWidth', 2) % Set all to same value
```
See Also

bar, plot, sort

“Area, Bar, and Pie Plots” for related functions

Area Graphs for more examples

“Areaseries Properties” for property descriptions
**Areaseries Properties**

**Modifying Properties**
You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

Note that you cannot define default properties for areaseries objects.

See Plot Objects for more information on areaseries objects.

**Areaseries Property Descriptions**
This section provides a description of properties. Curly braces `{ }` enclose default values.

**BaseValue**  double: \( y \)-axis value

*Location of filled area base.* You can specify the \( y \)-axis value where MATLAB draws the base of the filled area.

**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 areaseries 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 bar object.* The handle of a patch object that is the child of the areaseries 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 areaseries Children property unless you set the Root ShowHiddenHandles property to on:

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

**Clipping** {on} | off

*Clipping mode.* MATLAB clips area graphs to the axes plot box by default. If you set Clipping to off, areas can 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 an areaseries object. You must specify the callback during the creation of the object. For example,

```
area(y,'CreateFcn',@CallbackFcn)
```

where @CallbackFcn is a function handle that references the callback function.
MATLAB executes this routine after setting all other areaseries properties. Setting this property on an existing areaseries 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 areaseries object is deleted (e.g., this might happen when you issue a `delete` command on the areaseries 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 uses this text for labels for any areaseries objects appearing in these legends.

### `EdgeColor` {[0 0 0]} | none | `ColorSpec`

*Color of line that separates filled areas.* You can set the color of the edge of the filled areas to a three-element RGB vector or one of the MATLAB predefined names, including the string `none`. The default edge color is black. See `ColorSpec` for more information on specifying color.

### `EraseMode` {normal} | none | xor | background

*Erase mode.* This property controls the technique MATLAB uses to draw and erase areaseries child objects (the patch object used to construct the area 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.
Areaseries Properties

- normal — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- none — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with EraseMode none, you cannot print these objects because MATLAB stores no information about their former locations.

- xor — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes Color property is set to none). That is, it isn’t erased correctly if there are objects behind it.

- background — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes Color property is set to none). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Printing with Nonnormal Erase Modes

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

Set the axes background color with the axes Color property. Set the figure background color with the figure Color property.

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

FaceColor {flat} | none | ColorSpec

Color of filled areas. This property can be any of the following:
Areaseries Properties

- **ColorSpec** — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas. See ColorSpec for more information on specifying color.
- **none** — Do not draw faces. Note that **EdgeColor** is drawn independently of **FaceColor**.
- **flat** — The color of the filled areas is determined by the figure colormap. See **colormap** for information on setting the colormap.

**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 areaseries 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.
Areaseries 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 areaseries object can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click on the objects that compose the area graph. If HitTest is off, clicking the areaseries object selects the object below it (which is usually the axes containing it).

HitTestArea on | {off}
Select areaseries object on filled area or extent of graph. This property enables you to select areaseries objects in two ways:

• Select by clicking bars (default).
• Select by clicking anywhere in the extent of the area plot.

When HitTestArea is off, you must click the bars to select the bar object. When HitTestArea is on, you can select the bar object by clicking anywhere within the extent of the bar graph (i.e., anywhere within a rectangle that encloses all the bars).

Interruptible {on} | off
Callback routine interruption mode. The Interruptible property controls whether an areaseries 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 bar property. Note that MATLAB does not
Areaseries Properties

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 lines that separate filled areas. The following table shows 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>

**LineWidth**  
scalar

*The width of the line separating filled areas.* Specify this value in points (1 point = 1/72 inch). The default **LineWidth** is 0.5 points.

**Parent**  
axes handle

*Parent of areaseries object.* This property contains the handle of the areaseries object’s parent. The parent of an areaseries 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 areaseries object is selected.
Areaseries Properties

**SelectionHighlight**  
{on} | off  
*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

**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 an areaseries object and set the Tag property.

```plaintext
t = area(Y,'Tag','area1')
```

When you want to access the areaseries object, you can use `findobj` to find the areaseries object's handle. The following statement changes the FaceColor property of the object whose Tag is area1.

```plaintext
set(findobj('Tag','area1'),'FaceColor','red')
```

**Type**  
string (read only)  
*Type of graphics object.* This property contains a string that identifies the class of the graphics object. For areaseries objects, Type is 'hggroup'.

The following statement finds all the hggroup objects in the current axes.

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

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

**UserData**  
array  
*User-specified data.* This property can be any data you want to associate with the areaseries object (including cell arrays and structures). The areaseries
Areaseries Properties

object does not set values for this property, but you can access it using the set and get functions.

Visible  

| on | off |

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

XData  

A vector or matrix  

*The x-axis values for area graphs.* The x-axis values for area graphs are specified by the X input argument. If XData is a vector, \( \text{length}(\text{XData}) \) must equal \( \text{length}(\text{YData}) \) and must be monotonic. If XData is a matrix, \( \text{size}(\text{XData}) \) must equal \( \text{size}(\text{YData}) \) and each column must be monotonic.

XDataMode  

| auto | manual |

*Use automatic or user-specified x-axis values.* If you specify XData (by setting the XData property or specifying the x input argument), MATLAB sets this property to manual and uses the specified values to label the x-axis.

If you set XDataMode to auto after having specified XData, MATLAB resets the x-axis ticks to 1:size(YData,1).

XDataSource  

A string (MATLAB variable)  

*Link XData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

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

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

See the refreshdata reference page for more information.

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

**YData**

Vector or matrix

*Area plot data.* YData contains the data plotted as filled areas (the Y input argument). If YData is a vector, area creates a single filled area whose upper boundary is defined by the elements of YData. If YData is a matrix, area creates one filled area per column, stacking each on the previous plot.

The input argument Y in the area function calling syntax assigns values to YData.

**YDataSource**

String (MATLAB variable)

*Link YData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

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

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

See the refreshdata reference page for more information.

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

**Purpose**
Set FTP transfer type to ASCII.

**Syntax**
ascii(f)

**Description**
ascii(f) sets the download and upload FTP mode to ASCII, which converts new lines, where f was created using ftp. Use this function for text files only, including HTML pages and Rich Text Format (RTF) files.

**Examples**
Connect to The MathWorks FTP server, and display the FTP object.

```matlab
tmw=ftp('ftp.mathworks.com');
disp(tmw)
```  
**FTP Object**
- **host:** ftp.mathworks.com
- **user:** anonymous
- **dir:** /
- **mode:** binary

Note that the FTP object defaults to binary mode.

Use the ascii function to set the FTP mode to ASCII, and use the disp function to display the FTP object.

```matlab
ascii(tmw)
disp(tmw)
```  
**FTP Object**
- **host:** ftp.mathworks.com
- **user:** anonymous
- **dir:** /
- **mode:** ascii

Note that the FTP object is now set to ASCII mode.

**See Also**
ftp, binary (ftp)
Inverse secant, result in radians

Y = asec(X)

Y = asec(X) returns the inverse secant (arcsecant) for each element of X. The asec function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

Graph the inverse secant over the domains 1 ≤ x ≤ 5 and −5 ≤ x ≤ −1.

\[ x1 = -5:0.01:-1; \]
\[ x2 = 1:0.01:5; \]
\[ plot(x1,asec(x1),x2,asec(x2)), grid on \]

The inverse secant can be defined as

\[ \sec^{-1}(z) = \cos^{-1}\left(\frac{1}{z}\right) \]

asec 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.
asec

**See Also**
asecd, asech, sec
Purpose  Inverse secant, result in degrees
Syntax  \( Y = \text{asecd}(X) \)
Description  \( Y = \text{asecd}(X) \) is the inverse secant, expressed in degrees, of the elements of \( X \).
See Also  secd, asec
Purpose
Inverse hyperbolic secant

Syntax
Y = asech(X)

Description
Y = asech(X) returns the inverse hyperbolic secant for each element of X.
The asech function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples
Graph the inverse hyperbolic secant over the domain 0.01 ≤ x ≤ 1.

\[
x = 0.01:0.001:1;
\]

\[
plot(x,asech(x)), grid on
\]

Definition
The hyperbolic inverse secant can be defined as

\[
sech^{-1}(z) = \cosh^{-1}\left(\frac{1}{z}\right)
\]

Algorithm
asech 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, sech
Purpose
Inverse sine, result in radians

Syntax
Y = asin(X)

Description
Y = asin(X) returns the inverse sine (arcsine) for each element of X. For real elements of X in the domain \([-1, 1]\), asin(X) is in the range \([-\pi/2, \pi/2]\). For real elements of x outside the range \([-1, 1]\), asin(X) is complex.

The asin function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples
Graph the inverse sine function over the domain \(-1 \leq x \leq 1\).

\[
x = -1:.01:1;
\]

\[
plot(x, \text{asin}(x)), \text{ grid on}
\]

Definition
The inverse sine can be defined as

\[
\sin^{-1}(z) = -i \log \left( iz + (1 - z^2)^{1/2} \right)
\]
Algorithm

asin uses FDLIBM, which was developed at SunSoft, a Sun Microsystems, Inc. business, by Kwok C. Ng, and others. For information about FDLIBM, see http://www.netlib.org.

See Also

sin, asind, asinh
asind

**Purpose**  
Inverse sine, result in degrees

**Syntax**  
Y = asind(X)

**Description**  
Y = asind(X) is the inverse sine, expressed in degrees, of the elements of X.

**See Also**  
sind, asin
Purpose
Inverse hyperbolic sine

Syntax
Y = asinh(X)

Description
Y = asinh(X) returns the inverse hyperbolic sine for each element of X.
The asinh function operates element-wise on arrays. The function’s domains
and ranges include complex values. All angles are in radians.

Examples
Graph the inverse hyperbolic sine function over the domain \(-5 \leq x \leq 5\).

x = -5:.01:5;
plot(x,asinh(x)), grid on

Definition
The hyperbolic inverse sine can be defined as

\[
\sinh^{-1}(z) = \log\left[ z + \left( z^2 + 1 \right)^{\frac{1}{2}} \right]
\]
asinh 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, sinh
**Purpose**

Assign a value to a workspace variable

**Syntax**

assignin(ws, 'var', val)

**Description**

assignin(ws, 'var', val) assigns the value val to the variable var in the workspace ws. var is created if it doesn’t exist. ws can have a value of 'base' or 'caller' to denote the MATLAB base workspace or the workspace of the caller function.

The assignin function is particularly useful for these tasks:

- Exporting data from a function to the MATLAB workspace
- Within a function, changing the value of a variable that is defined in the workspace of the caller function (such as a variable in the function argument list)

**Remarks**

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the M-file. Note that the base and caller workspaces are equivalent in the context of an M-file that is invoked from the MATLAB command line.

**Examples**

This example creates a dialog box for the image display function, prompting a user for an image name and a colormap name. The assignin function is used to export the user-entered values to the MATLAB workspace variables imfile and cmap.

```matlab
prompt = {'Enter image name:','Enter colormap name:'};
title = 'Image display - assignin example';
lines = 1;
def = {'my_image','hsv'};
answer = inputdlg(prompt,title,lines,def);
assignin('base','imfile',answer{1});
assignin('base','cmap',answer{2});
```
assignin

See Also evalin
Purpose
Inverse tangent, result in radians

Syntax
\[ Y = \text{atan}(X) \]

Description
\( Y = \text{atan}(X) \) returns the inverse tangent (arctangent) for each element of \( X \).
For real elements of \( X \), \( \text{atan}(X) \) is in the range \([-\pi/2, \pi/2]\).

The \( \text{atan} \) function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples
Graph the inverse tangent function over the domain \(-20 \leq x \leq 20\).

\[
x = -20:0.01:20;
plot(x,atan(x)), grid on
\]

Definition
The inverse tangent can be defined as
\[
\tan^{-1}(z) = \frac{i}{2} \log\left(\frac{i + z}{i - z}\right)
\]
atan 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  
atan2, tan, atand, atanh
atan2

**Purpose**
Four-quadrant inverse tangent

**Syntax**
P = atan2(Y,X)

**Description**
P = atan2(Y,X) returns an array P the same size as X and Y containing the element-by-element, four-quadrant inverse tangent (arctangent) of the real parts of Y and X. Any imaginary parts are ignored.

Elements of P lie in the closed interval [-pi,pi], where pi is the MATLAB floating-point representation of \( \pi \). atan uses \( \text{sign}(Y) \) and \( \text{sign}(X) \) to determine the specific quadrant.

atan2(Y,X) contrasts with atan(Y/X), whose results are limited to the interval \([-\pi/2, \pi/2]\), or the right side of this diagram.

**Examples**
Any complex number \( z = x + iy \) is converted to polar coordinates with

\[
\begin{align*}
    r &= \text{abs}(z) \\
    \theta &= \text{atan2}(\text{imag}(z),\text{real}(z))
\end{align*}
\]

For example,

\[
\begin{align*}
    z &= 4 + 3i; \\
    r &= \text{abs}(z) \\
    \theta &= \text{atan2}(\text{imag}(z),\text{real}(z))
\end{align*}
\]

\[
\begin{align*}
    r &= 5 \\
    \theta &= 0.6435
\end{align*}
\]
This is a common operation, so MATLAB provides a function, angle(z), that computes \( \theta = \text{atan2} \left( \text{imag}(z), \text{real}(z) \right) \).

To convert back to the original complex number

\[
z = r \times \exp(i \times \theta)
\]

\[
z = 4.0000 + 3.0000i
\]

**Algorithm**

atan2 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**

angle, atan, atanh
Purpose  
Inverse tangent, result in degrees

Syntax  
$Y = \text{atand}(X)$

Description  
$Y = \text{atand}(X)$ is the inverse tangent, expressed in degrees, of the elements of $X$.

See Also  
tand, atan
atanh

Purpose
Inverse hyperbolic tangent

Syntax
Y = atanh(X)

Description
The atanh function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Y = atanh(X) returns the inverse hyperbolic tangent for each element of X.

Examples
Graph the inverse hyperbolic tangent function over the domain \(-1 < x < 1\).

\[
x = -0.99:0.01:0.99;
\]

\[
plot(x,atanh(x)), grid on
\]

Definition
The hyperbolic inverse tangent can be defined as

\[
\tanh^{-1}(z) = \frac{1}{2} \log\left(\frac{1 + z}{1 - z}\right)
\]

Algorithm
atanh 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

atan2, atan, tanh
**audioplayer**

**Purpose**: Create an audio player object

**Syntax**

```matlab
y = audioplayer(x,Fs)
y = audioplayer(x,Fs,nbits)
y = audioplayer(r)
y = audioplayer(r,id)
```

**Description**

**Note** To use all of the features of the audio player object, your system needs a properly installed and configured sound card with 8- and 16-bit I/O, two channels, and support for sampling rates of up to 48 kHz.

*y = audioplayer(x,Fs)* returns a handle to an audio player object *y* using input audio signal *x*. The audio player object supports methods and properties that you can use to play audio data.

The input signal *x* can be a vector or two-dimensional array containing `single`, `double`, `int8`, `uint8`, or `int16` MATLAB data types. The input sample value range depends on the MATLAB data type.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Input Sample Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int8</code></td>
<td>-128 to 127</td>
</tr>
<tr>
<td><code>uint8</code></td>
<td>0 to 255</td>
</tr>
<tr>
<td><code>int16</code></td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td><code>single</code></td>
<td>-1 to 1</td>
</tr>
<tr>
<td><code>double</code></td>
<td>-1 to 1</td>
</tr>
</tbody>
</table>

*Fs* is the sampling rate in Hz to use for playback. Valid values for *Fs* depend on the specific audio hardware installed. Typical values supported by most sound cards are 8000, 11025, 22050, and 44100 Hz.

*y = audioplayer(x,Fs,nbits)* returns a handle to an audio player object where *nbits* is the bit quantization to use for `single` or `double` data types. This is an optional parameter with a default value of 16. Valid values for *nbits* are 8 and 16 (and 24, if a 24-bit device is installed). You do not need to specify *nbits*.
for int8, uint8, or int16 data because the quantization is set automatically to 8 or 16, respectively.

\[ y = \text{audioplayer}(r) \]
returns a handle to an audio player object from an audiorecorder object \( r \).

\[ y = \text{audioplayer}(r, \text{id}) \]
returns a handle to an audio player object from an audiorecorder object \( r \), using the audio device specified by \( \text{id} \) for output. This option is only available on systems running Windows.

**Example**

Load a sample audio file, create an audio player object, and play the audio at a higher sampling rate. \( x \) contains the audio samples and \( F_s \) is the sampling rate. You can use any of the audioplayer functions listed above on the player.

```matlab
load handel;
player = audioplayer(y,Fs);
play(player,[1 (get(player,'SampleRate')*3)]);
```

To stop the playback, use this command:

```matlab
stop(player);  % Equivalent to player.stop
```

**Methods**

After you create an audio player object, you can use the methods listed below on that object. \( y \) represents the name of the returned audio player.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>play(y)</code></td>
<td>Starts playback from the beginning and plays to the end, or from start sample to the end, or from start sample to stop sample. The values of start and stop can be specified in a two-element vector range.</td>
</tr>
<tr>
<td><code>play(y,start)</code></td>
<td></td>
</tr>
<tr>
<td><code>play(y,range)</code></td>
<td></td>
</tr>
<tr>
<td><code>playblocking(y)</code></td>
<td>Same as play, but does not return control until playback completes.</td>
</tr>
<tr>
<td><code>playblocking(y,start)</code></td>
<td></td>
</tr>
<tr>
<td><code>playblocking(y,range)</code></td>
<td></td>
</tr>
<tr>
<td><code>stop(y)</code></td>
<td>Stops playback.</td>
</tr>
</tbody>
</table>
**audioplayer**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pause(y)</td>
<td>Pauses playback.</td>
</tr>
<tr>
<td>resume(y)</td>
<td>Restarts playback from where playback was paused.</td>
</tr>
<tr>
<td>isplaying(y)</td>
<td>Indicates whether playback is in progress. If 0, playback is not in progress. If 1, playback is in progress.</td>
</tr>
<tr>
<td>disp(y)</td>
<td>Displays all property information about audio player y.</td>
</tr>
<tr>
<td>get(y)</td>
<td></td>
</tr>
</tbody>
</table>

### Properties

Audio player objects have the properties listed below. To set a user-settable property, use this syntax:

```matlab
set(y, 'property1', value,'property2',value,...)
```

To view a read-only property,

```matlab
get(y,'property') % Displays 'property' setting.
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Name of the object’s class</td>
<td>Read-only</td>
</tr>
<tr>
<td>SampleRate</td>
<td>Sampling frequency in Hz</td>
<td>User-settable</td>
</tr>
<tr>
<td>BitsPerSample</td>
<td>Number of bits per sample</td>
<td>Read-only</td>
</tr>
<tr>
<td>NumberOfChannels</td>
<td>Number of channels</td>
<td>Read-only</td>
</tr>
<tr>
<td>TotalSamples</td>
<td>Total length, in samples, of the audio data</td>
<td>Read-only</td>
</tr>
<tr>
<td>Running</td>
<td>Status of the audio player (‘on’ or ‘off’)</td>
<td>Read-only</td>
</tr>
</tbody>
</table>
### audioplayer

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentSample</td>
<td>Current sample being played by the audio output device (if it is not playing, currentSample is the next sample to be played with play or resume)</td>
<td>Read-only</td>
</tr>
<tr>
<td>UserData</td>
<td>User data of any type</td>
<td>User-settable</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object label string</td>
<td>User-settable</td>
</tr>
</tbody>
</table>

For information on using the following four properties, see Creating Timer Callback Functions in the MATLAB documentation. Note that for audio object callbacks, `eventStruct (event)` is currently empty (`[]`).

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerFcn</td>
<td>Name of or handle to user-specified function to be called during playback</td>
<td>User-settable</td>
</tr>
<tr>
<td>TimerPeriod</td>
<td>Time, in seconds, between TimerFcn callbacks</td>
<td>User-settable</td>
</tr>
<tr>
<td>StartFcn</td>
<td>Name of or handle to the function to be called once when playback starts</td>
<td>User-settable</td>
</tr>
<tr>
<td>StopFcn</td>
<td>Name of or handle to the function to be called once when playback stops</td>
<td>User-settable</td>
</tr>
</tbody>
</table>

**See Also**

audiorecorder, sound, wavplay, wavwrite, wavread, get, set, methods
audiorecorder

Purpose
Create an audio recorder object

Syntax
```
y = audiorecorder
y = audiorecorder(Fs,nbits,channels)
y = audiorecorder(Fs,nbits,channels,id)
```

Description

**Note** To use all of the features of the audio recorder object, your system must have a properly installed and configured sound card with 8- and 16-bit I/O and support for sampling rates of up to 48 kHz.

- `y = audiorecorder` returns a handle to an 8-kHz, 8-bit, mono audio recorder object. The audio recorder object supports methods and properties that you can use to record audio data.

- `y = audiorecorder(Fs,nbits,channels)` returns a handle to an audio recorder object using the sampling rate `Fs` (in Hz), the sample size of `nbits`, and the number of `channels`. `Fs` can be any sampling rate supported by the audio hardware. Common sampling rates are 8000, 11025, 22050, and 44000. The value of `nbits` must be 8 or 16 (or 24, if a 24-bit device is installed). For mono or stereo, `channels` must be 1 or 2, respectively.

- `y = audiorecorder(Fs,nbits,channels,id)` returns a handle to an audio recorder object using the audio device specified by its `id` for input.

Examples

**Example 1**
Using a microphone, record 3.5 seconds of 44.1-kHz, 16-bit, stereo data, and then return the data to the MATLAB workspace as a double array.
```
recorder = audiorecorder(44100,16,2);
recordblocking(recorder,3.5);
audioarray = getaudiodata(recorder);
```

**Example 2**
Using a microphone, record 8-bit, 22-kHz mono data, play it back, record again, and return the data to the MATLAB workspace as a `uint8` array.
```
micrecorder = audiorecorder(22050,8,1);
record(micrecorder);
```
% Now, speak into microphone

stop(micrecorder);
speechplayer = play(micrecorder);
% Now, listen to the recording

stop(speechplayer);
speechdata = getaudiodata(micrecorder, 'uint8');

Remarks
The current implementation of audiorecorder is not intended for long,
high-sample-rate recording because it uses system memory for storage and
does not use disk buffering. When large recordings are attempted, MATLAB
performance may degrade.

Methods
After you create an audio recorder object, you can use the methods listed below
on that object. y represents the name of the returned audio recorder.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>record(y)</td>
<td>Starts recording.</td>
</tr>
<tr>
<td>record(y,length)</td>
<td>Records for length number of seconds.</td>
</tr>
<tr>
<td>recordblocking(y,length)</td>
<td>Same as record, but does not return control until recording completes.</td>
</tr>
<tr>
<td>stop(y)</td>
<td>Stops recording.</td>
</tr>
<tr>
<td>pause(y)</td>
<td>Pauses recording.</td>
</tr>
<tr>
<td>resume(y)</td>
<td>Restarts recording from where recording was paused.</td>
</tr>
</tbody>
</table>
| isrecording(y)   | Indicates the status of recording. If 0, recording is not in progress. If 1,
                  | recording is in progress.                                                   |
| play(y)          | Creates an audioplayer, plays the recorded audio data, and returns a handle to the created audioplayer. |
Audio recorder objects have the properties listed below. To set a user-settable property, use this syntax:

```
set(y, 'property1', value,'property2',value,...)
```

To view a read-only property,

```
get(y,'property') %displays 'property' setting.
```

### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getplayer(y)</td>
<td>Creates an audioplayer and returns a handle to the created audioplayer.</td>
</tr>
<tr>
<td>getaudiodata(y)</td>
<td>Returns the recorded audio data to the MATLAB workspace. type is a string containing the desired data type. Supported data types are double, single, int16, int8, or uint8. If type is omitted, it defaults to 'double'. For double and single, the array contains values between -1 and 1. For int8, values are between -128 to 127. For uint8, values are from 0 to 255. For int16, values are from -32768 to 32767. If the recording is in mono, the returned array has one column. If it is in stereo, the array has two columns, one for each channel.</td>
</tr>
<tr>
<td>display(y)</td>
<td>Displays all property information about audio recorder y.</td>
</tr>
<tr>
<td>disp(y)</td>
<td></td>
</tr>
<tr>
<td>get(y)</td>
<td></td>
</tr>
</tbody>
</table>

### Properties

Audio recorder objects have the properties listed below. To set a user-settable property, use this syntax:

```
set(y, 'property1', value,'property2',value,...)
```

To view a read-only property,

```
get(y,'property') %displays 'property' setting.
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Name of the object's class</td>
<td>Read-only</td>
</tr>
<tr>
<td>SampleRate</td>
<td>Sampling frequency in Hz</td>
<td>Read-only</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>BitsPerSample</td>
<td>Number of bits per recorded sample</td>
<td>Read-only</td>
</tr>
<tr>
<td>NumberOfChannels</td>
<td>Number of channels of recorded audio</td>
<td>Read-only</td>
</tr>
<tr>
<td>TotalSamples</td>
<td>Total length, in samples, of the recording</td>
<td>Read-only</td>
</tr>
<tr>
<td>Running</td>
<td>Status of the audio recorder ('on' or 'off')</td>
<td>Read-only</td>
</tr>
<tr>
<td>CurrentSample</td>
<td>Current sample being recorded by the audio output device (if it is not recording, current sample is the next sample to be recorded with record or resume)</td>
<td>Read-only</td>
</tr>
<tr>
<td>UserData</td>
<td>User data of any type</td>
<td>User-settable</td>
</tr>
</tbody>
</table>

For information on using the following four properties, see Creating Timer Callback Functions in the MATLAB documentation. Note that for audio object callbacks, eventStruct (event) is currently empty ([]).

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerFcn</td>
<td>Name of or handle to user-specified function to be called during recording</td>
<td>User-settable</td>
</tr>
<tr>
<td>TimerPeriod</td>
<td>Time, in seconds, between TimerFcn callbacks</td>
<td>User-settable</td>
</tr>
<tr>
<td>StartFcn</td>
<td>Name of or handle to the function to be called a single time when recording starts</td>
<td>User-settable</td>
</tr>
<tr>
<td>StopFcn</td>
<td>Name of or handle to the function to be called a single time when recording stops</td>
<td>User-settable</td>
</tr>
</tbody>
</table>
## audiorecorder

### Data Structure

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumberOfBuffers</td>
<td>Number of buffers used for recording (you should adjust this only if you have skips, dropouts, etc., in your recording)</td>
<td>User-settable</td>
</tr>
<tr>
<td>BufferLength</td>
<td>Length in seconds of buffer (you should adjust this only if you have skips, dropouts, etc., in your recording)</td>
<td>User-settable</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified object label string</td>
<td>User-settable</td>
</tr>
</tbody>
</table>

### See Also

audioplayer, wavread, wavrecord, wavwrite, get, set, methods
Purpose
Return information about the NeXT/SUN (.au) sound file

Syntax
[m d] = aufinfo(aufile)

Description
[m d] = aufinfo(aufile) returns information about the contents of the AU sound file specified by the string aufile.

m is the string 'Sound (AU) file', if filename is an AU 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 an AU file, it contains the string 'Not an AU file'.

See Also
auread
Read NeXT/SUN (.au) sound file

As an alternative to `auread`, use the Import Wizard. To activate the Import Wizard, select Import data from the File menu.

`y = auread('aufile')`

`[y,Fs,bits] = auread('aufile')`

`[...] = auread('aufile',N)`

`[...] = auread('aufile',N1,N2)`

`siz = auread('aufile','size')`

`y = auread('aufile')` loads a sound file specified by the string `aufile`, returning the sampled data in `y`. The .au extension is appended if no extension is given. Amplitude values are in the range [-1,+1]. `auread` supports multichannel data in the following formats:

- 8-bit mu-law
- 8-, 16-, and 32-bit linear
- Floating-point

`[y,Fs,bits] = auread('aufile')` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.

`[...] = auread('aufile',N)` returns only the first `N` samples from each channel in the file.

`[...] = auread('aufile',N1,N2)` returns only samples `N1` through `N2` from each channel in the file.

`siz = auread('aufile','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: `auwrite`, `wavread`
**Purpose**
Write NeXT/SUN (.au) sound file

**Syntax**

auwrite(y,'aufile')
auwrite(y,Fs,'aufile')
auwrite(y,Fs,N,'aufile')
auwrite(y,Fs,N,'method','aufile')

**Description**

auwrite(y,'aufile') writes a sound file specified by the string aufile. The data should be arranged with one channel per column. Amplitude values outside the range [-1,+1] are clipped prior to writing. auwrite supports multichannel data for 8-bit mu-law and 8- and 16-bit linear formats.

auwrite(y,Fs,'aufile') specifies the sample rate of the data in Hertz.

auwrite(y,Fs,N,'aufile') selects the number of bits in the encoder. Allowable settings are N = 8 and N = 16.

auwrite(y,Fs,N,'method','aufile') allows selection of the encoding method, which can be either mu or linear. Note that mu-law files must be 8-bit. By default, method = 'mu'.

**See Also**
auread, wavwrite
**Purpose**

Create a new Audio/Video Interleaved (AVI) file

**Syntax**

```matlab
aviobj = avifile(filename)

aviobj = avifile(filename,'PropertyName',value,'PropertyName',value,...)
```

**Description**

`aviobj = avifile(filename)` creates an AVI file, giving it the name specified in `filename`, using default values for all AVI file object properties. If `filename` does not include an extension, `avifile` appends `.avi` to the filename. AVI is a file format for storing audio and video data.

`avifile` returns a handle to an AVI file object `aviobj`. You use this object to refer to the AVI file in other functions. An AVI file object supports properties and methods that control aspects of the AVI file created.

`aviobj = avifile(filename,'Param',Value,'Param',Value,...)` creates an AVI file with the specified parameter settings. This table lists available parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>'colormap'</td>
<td>An m-by-3 matrix defining the colormap to be used for indexed AVI movies, where m must be no greater than 256 (236 if using Indeo compression). You must set this parameter before calling <code>addframe</code>, unless you are using <code>addframe</code> with the MATLAB movie syntax.</td>
<td>There is no default colormap.</td>
</tr>
<tr>
<td>'compression'</td>
<td>A text string specifying the compression codec to use.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>On Windows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'Indeo3'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'Indeo5'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'Cinepak'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'MSVC'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'None'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>On UNIX:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'None'</td>
<td>'Indeo5' on Windows.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'None' on UNIX.</td>
</tr>
</tbody>
</table>
You can also use structure syntax to set AVI file object properties. For example, to set the quality property to 100, use the following syntax:

```matlab
aviobj = avifile('myavifile');
aviobj.Quality = 100;
```

### Example

This example shows how to use the `avifile` function to create the AVI file `example.avi`.

```matlab
fig=figure;
set(fig,'DoubleBuffer','on');
set(gca,'xlim',[-80 80],'ylim',[-80 80],...
    'NextPlot','replace','Visible','off')
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>'fps'</td>
<td>A scalar value specifying the speed of the AVI movie in frames per second (fps).</td>
<td>15 fps</td>
</tr>
<tr>
<td>'keyframe'</td>
<td>For compressors that support temporal compression, this is the number of key frames per second.</td>
<td>2 key frames per second.</td>
</tr>
<tr>
<td>'quality'</td>
<td>A number between 0 and 100. This parameter has no effect on uncompressed movies. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes.</td>
<td>75</td>
</tr>
<tr>
<td>'videoname'</td>
<td>A descriptive name for the video stream. This parameter must be no greater than 64 characters long.</td>
<td>The default is the filename.</td>
</tr>
</tbody>
</table>

To use a custom compression codec, specify the four-character code that identifies the codec (typically included in the codec documentation). The `addframe` function reports an error if it cannot find the specified custom compressor.
mov = avifile('example.avi')
x = -pi:.1:pi;
radius = 0:length(x);
for k=1:length(x)
    h = patch(sin(x)*radius(k),cos(x)*radius(k),...
        [abs(cos(x(k))) 0 0]);
    set(h,'EraseMode','xor');
    F = getframe(gca);
    mov = addframe(mov,F);
end
mov = close(mov);

See Also
addframe, close, movie2avi
**Purpose**

Return information about an Audio/Video Interleaved (AVI) file

**Syntax**

`fileinfo = aviinfo(filename)`

**Description**

`fileinfo = aviinfo(filename)` returns a structure whose fields contain information about the AVI file specified in the string `filename`. If `filename` does not include an extension, then `.avi` is used. The file must be in the current working directory or in a directory on the MATLAB path.

The set of fields in the `fileinfo` structure is shown below.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioFormat</td>
<td>String containing the name of the format used to store the audio data, if audio data is present</td>
</tr>
<tr>
<td>AudioRate</td>
<td>Integer indicating the sample rate in Hertz of the audio stream, if audio data is present</td>
</tr>
<tr>
<td>Filename</td>
<td>String specifying the name of the file</td>
</tr>
<tr>
<td>FileModDate</td>
<td>String containing the modification date of the file</td>
</tr>
<tr>
<td>FileSize</td>
<td>Integer indicating the size of the file in bytes</td>
</tr>
<tr>
<td>FramesPerSecond</td>
<td>Integer indicating the desired frames per second</td>
</tr>
<tr>
<td>Height</td>
<td>Integer indicating the height of the AVI movie in pixels</td>
</tr>
<tr>
<td>ImageType</td>
<td>String indicating the type of image. Either ‘truecolor’ for a truecolor (RGB) image, or ‘indexed’ for an indexed image.</td>
</tr>
<tr>
<td>NumAudioChannels</td>
<td>Integer indicating the number of channels in the audio stream, if audio data is present</td>
</tr>
<tr>
<td>NumFrames</td>
<td>Integer indicating the total number of frames in the movie</td>
</tr>
<tr>
<td>NumColormapEntries</td>
<td>Integer specifying the number of colormap entries. For a truecolor image, this value is 0 (zero).</td>
</tr>
</tbody>
</table>
**Quality**
Number between 0 and 100 indicating the video quality in the AVI file. Higher quality numbers indicate higher video quality; lower quality numbers indicate lower video quality. This value is not always set in AVI files and therefore can be inaccurate.

**VideoCompression**
String containing the compressor used to compress the AVI file. If the compressor is not Microsoft Video 1, Run Length Encoding (RLE), Cinepak, or Intel Indeo, *aviinfo* returns the four-character code that identifies the compressor.

**Width**
Integer indicating the width of the AVI movie in pixels.

### See also
*avifile, aviread*
Purpose
Read an Audio/Video Interleaved (AVI) file

Syntax
\[ \text{mov} = \text{aviread}(\text{filename}) \]
\[ \text{mov} = \text{aviread}(\text{filename}, \text{index}) \]

Description
\[ \text{mov} = \text{aviread}(\text{filename}) \] reads the AVI movie \text{filename} into the MATLAB movie structure \text{mov}. If \text{filename} does not include an extension, then .avi is used. Use the \text{movie} function to view the movie \text{mov}. On UNIX, \text{filename} must be an uncompressed AVI file.

\text{mov} has two fields, \text{cdata} and \text{colormap}. The content of these fields varies depending on the type of image.

<table>
<thead>
<tr>
<th>Image Type</th>
<th>cdata Field</th>
<th>colormap Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truecolor</td>
<td>Height-by-width-by-3 array</td>
<td>Empty</td>
</tr>
<tr>
<td>Indexed</td>
<td>Height-by-width array</td>
<td>( m )-by-3 array</td>
</tr>
</tbody>
</table>

The supported frame types are 8-bit, for indexed or grayscale images, 16-bit, for grayscale images, or 24-bit, for truecolor.

\[ \text{mov} = \text{aviread}(\text{filename}, \text{index}) \] reads only the frames specified by \text{index}. \text{index} can be a single index or an array of indices into the video stream. In AVI files, the first frame has the index value 1, the second frame has the index value 2, and so on.

See also
\text{aviinfo}, \text{avifile}, \text{movie}
Purpose

Create axes graphics object

Syntax

axes
axes('PropertyName',PropertyValue,...)
axes(h)
h = axes(...)

Description

axes is the low-level function for creating axes graphics objects.

axes creates an axes graphics object in the current figure using default property values.

axes('PropertyName',PropertyValue,...) creates an axes object having the specified property values. MATLAB uses default values for any properties that you do not explicitly define as arguments.

axes(h) makes existing axes h the current axes. It also makes h the first axes listed in the figure’s Children property and sets the figure’s CurrentAxes property to h. The current axes is the target for functions that draw image, line, patch, surface, and text graphics objects.

h = axes(...) returns the handle of the created axes object.

Remarks

MATLAB automatically creates an axes, if one does not already exist, when you issue a command that creates a graph.

The axes function accepts property name/property value pairs, structure arrays, and cell arrays as input arguments (see the set and get commands for examples of how to specify these data types). These properties, which control various aspects of the axes object, are described in the “Axes Properties” section.

Use the set function to modify the properties of an existing axes or the get function to query the current values of axes properties. Use the gca command to obtain the handle of the current axes.

The axis (not axes) function provides simplified access to commonly used properties that control the scaling and appearance of axes.
While the basic purpose of an axes object is to provide a coordinate system for plotted data, axes properties provide considerable control over the way MATLAB displays data.

**Stretch-to-Fill**
By default, MATLAB stretches the axes to fill the axes position rectangle (the rectangle defined by the last two elements in the Position property). This results in graphs that use the available space in the rectangle. However, some 3-D graphs (such as a sphere) appear distorted because of this stretching, and are better viewed with a specific three-dimensional aspect ratio.

Stretch-to-fill is active when the DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all auto (the default). However, stretch-to-fill is turned off when the DataAspectRatio, PlotBoxAspectRatio, or CameraViewAngle is user-specified, or when one or more of the corresponding modes is set to manual (which happens automatically when you set the corresponding property value).

This picture shows the same sphere displayed both with and without the stretch-to-fill. The dotted lines show the axes rectangle.

When stretch-to-fill is disabled, MATLAB sets the size of the axes to be as large as possible within the constraints imposed by the Position rectangle without
introducing distortion. In the picture above, the height of the rectangle constrains the axes size.

**Examples**

**Zooming**

Zoom in using aspect ratio and limits:

```matlab
sphere
set(gca,'DataAspectRatio',[1 1 1],...
    'PlotBoxAspectRatio',[1 1 1],'ZLim',[-0.6 0.6])
```

Zoom in and out using the `CameraViewAngle`:

```matlab
sphere
set(gca,'CameraViewAngle',get(gca,'CameraViewAngle')-5)
set(gca,'CameraViewAngle',get(gca,'CameraViewAngle')+5)
```

Note that both examples disable the MATLAB stretch-to-fill behavior.

**Positioning the Axes**

The `axes` `Position` property enables you to define the location of the axes within the figure window. For example,

```matlab
h = axes('Position',position_rectangle)
```

creates an axes object at the specified position within the current figure and returns a handle to it. Specify the location and size of the axes with a rectangle defined by a four-element vector,

```matlab
position_rectangle = [left, bottom, width, height];
```

The `left` and `bottom` elements of this vector define the distance from the lower left corner of the figure to the lower left corner of the rectangle. The `width` and `height` elements define the dimensions of the rectangle. You specify these values in units determined by the `Units` property. By default, MATLAB uses normalized units where (0,0) is the lower left corner and (1.0,1.0) is the upper right corner of the figure window.

You can define multiple axes in a single figure window:

```matlab
axes('position', [.1 .1 .8 .6])
mesh(peaks(20));
axes('position', [.1 .7 .8 .2])
pcolor([1:10;1:10]);
```
In this example, the first plot occupies the bottom two-thirds of the figure, and the second occupies the top third.

See Also

axis, cla, clf, figure, gca, grid, subplot, title, xlabel, ylabel, zlabel, view

“Axes Operations” for related functions

Axes Properties for more examples

See Types of Graphics Objects for information on core, group, plot, and annotation objects.
You can set default axes properties on the figure and root levels:

```matlab
set(0,'DefaultAxesPropertyName',PropertyValue,...)
set(gcf,'DefaultAxesPropertyName',PropertyValue,...)
```

where *PropertyName* is the name of the axes property and *PropertyValue* is the value you are specifying. Use `set` and `get` to access axes properties.

The following table lists all axes 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>Controlling Style and Appearance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Box</td>
<td>Toggles axes plot box on and off</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
<tr>
<td>Clipping</td>
<td>This property has no effect; axes are always clipped to the figure window.</td>
<td></td>
</tr>
<tr>
<td>GridLineStyle</td>
<td>Line style used to draw axes grid lines</td>
<td>Values: -, --, :, -.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: : (dotted line)</td>
</tr>
<tr>
<td>MinorGridLineStyle</td>
<td>Line style used to draw axes minor grid lines</td>
<td>Values: -, --, :, -.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: : (dotted line)</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>Layer</td>
<td>Draws axes above or below graphs</td>
<td>Values: bottom, top</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: bottom</td>
</tr>
<tr>
<td>LineStyleOrder</td>
<td>Sequence of line styles used for multiline plots</td>
<td>Values: LineSpec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: – (solid line for)</td>
</tr>
<tr>
<td>LineWidth</td>
<td>Width of axis lines, in points (1/72&quot; per point)</td>
<td>Values: number of points</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: 0.5 points</td>
</tr>
<tr>
<td>SelectionHighlight</td>
<td>Highlights axes when selected (Selected property set to on)</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>TickDir</td>
<td>Direction of axis tick marks</td>
<td>Values: in, out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: in (2-D), out (3-D)</td>
</tr>
<tr>
<td>TickDirMode</td>
<td>Use MATLAB or user-specified tick mark direction</td>
<td>Values: auto, manual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: auto</td>
</tr>
<tr>
<td>TickLength</td>
<td>Length of tick marks normalized to axis line length,</td>
<td>Values: [2-D 3-D]</td>
</tr>
<tr>
<td></td>
<td>specified as two-element vector</td>
<td>Default: [0.01 0.025]</td>
</tr>
<tr>
<td>Visible</td>
<td>Make axes visible or invisible</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: on</td>
</tr>
<tr>
<td>XGrid, YGrid, ZGrid</td>
<td>Toggle grid lines on and off in respective axis</td>
<td>Values: on, off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: off</td>
</tr>
</tbody>
</table>

### General Information About the Axes

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivePositionProperty</td>
<td>Determines whether the OuterPosition or Position property determines size of axes after resize</td>
<td>Values: outerposition, position</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: outerposition</td>
</tr>
<tr>
<td>Children</td>
<td>Handles of the images, lights, lines, patches, surfaces, and text objects displayed in the axes</td>
<td>Value: vector of handles</td>
</tr>
<tr>
<td>CurrentPoint</td>
<td>Location of last mouse button click defined in the axes data units</td>
<td>Value: a 2-by-3 matrix</td>
</tr>
<tr>
<td><strong>Property Name</strong></td>
<td><strong>Property Description</strong></td>
<td><strong>Property Value</strong></td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>HitTest</td>
<td>Specifies whether axes can become the current object (see figure CurrentObject property)</td>
<td>Values: on, off Default: on</td>
</tr>
<tr>
<td>OuterPosition</td>
<td>Position of axes including axis labels, title, and a margin</td>
<td>Value: [left bottom width height] Default: [0 0 1 1] in normalized units</td>
</tr>
<tr>
<td>Parent</td>
<td>Handle of the figure or uipanel containing the axes</td>
<td>Values: scalar figure or uipanel handle</td>
</tr>
<tr>
<td>Position</td>
<td>Location and size of axes within the figure</td>
<td>Values: [left bottom width height] Default: [0.1300 0.1100 0.7750 0.8150] in normalized Units</td>
</tr>
<tr>
<td>TightInset</td>
<td>Margin added to Position to include labels and title</td>
<td>Values: [left, bottom, right, top] Read only</td>
</tr>
<tr>
<td>Selected</td>
<td>Indicates whether axes is in a selected state</td>
<td>Values: on, off Default: on</td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified label</td>
<td>Values: any string Default: ‘ ’ (empty string)</td>
</tr>
<tr>
<td>Type</td>
<td>The type of graphics object (read only)</td>
<td>Value: the string 'axes'</td>
</tr>
<tr>
<td>Units</td>
<td>Units used to interpret the Position property</td>
<td>Values: inches, centimeters, characters, normalized, points, pixels Default: normalized</td>
</tr>
<tr>
<td>UserData</td>
<td>User-specified data</td>
<td>Value: any matrix Default: [] (empty matrix)</td>
</tr>
</tbody>
</table>

**Selecting Fonts and Labels**
### Property Name | Property Description | Property Value
--- | --- | ---
FontAngle | Selects italic or normal font | Values: normal, italic, oblique  Default: normal
FontName | Font family name (e.g., Helvetica, Courier) | Values: a font supported by your system or the string  FixedWidth  Default: typically Helvetica
FontSize | Size of the font used for title and labels | Value: an integer in FontUnits  Default: 10
FontUnits | Units used to interpret the FontSize property | Values: points, normalized, inches, centimeters, pixels  Default: points
FontWeight | Selects bold or normal font | Values: normal, bold, light, demi  Default: normal
Title | Handle of the title text object | Value: any valid text object handle
XLabel, YLabel, ZLabel | Handles of the respective axis label text objects | Value: any valid text object handle
XTickLabel, YTickLabel, ZTickLabel | Specifies tick mark labels for the respective axis | Value: matrix of strings  Defaults: numeric values selected automatically by MATLAB
XTickLabelMode, YTickLabelMode, ZTickLabelMode | Uses MATLAB or user-specified tick mark labels | Values: auto, manual  Default: auto

### Controlling Axis Scaling
### axes

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| XAxisLocation       | Specifies the location of the x-axis             | Values: top, bottom  
                        | Default: bottom                                                             |
| YAxisLocation       | Specifies the location of the y-axis             | Values: right, left  
                        | Default: left                                                              |
| XDir, YDir, ZDir    | Specifies the direction of increasing            | Values: normal, reverse  
                        | Default: normal                                                             |
| XLim, YLim, ZLim    | Specifies the limits to the respective axes      | Values: [min max]  
                        | Default: min and max determined automatically by MATLAB                     |
| XLimMode, YLimMode, ZLimMode | Uses MATLAB or user-specified values for the respective axis limits | Values: auto, manual  
                        | Default: auto                                                               |
| XMinorGrid, YMinorGrid, ZMinorGrid | Determines whether MATLAB displays gridlines connecting minor tick marks in the respective axis | Values: on, off  
                        | Default: off                                                                |
| XMinorTick, YMinorTick, ZMinorTick | Determines whether MATLAB displays minor tick marks in the respective axis | Values: on, off  
                        | Default: off                                                                |
| XScale, YScale, ZScale | Selects linear or logarithmic scaling of the respective axis | Values: linear, log  
                        | Default: linear (changed by plotting commands that create nonlinear plots)    |
| XTick, YTick, ZTick | Specifies the location of the axis tick marks    | Values: a vector of data values locating tick marks  
                        | Default: MATLAB automatically determines tick mark placement                 |
| XTickMode, YTickMode, ZTickMode | Uses MATLAB or user-specified values for the respective tick mark locations | Values: auto, manual  
                        | Default: auto                                                                |
### Controlling the View

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| CameraPosition        | Specifies the position of the point from which you view the scene | Values: \([x, y, z]\) axes coordinates  
Default: automatically determined by MATLAB |
| CameraPositionMode    | Uses MATLAB or user-specified camera position | Values: auto, manual  
Default: auto |
| CameraTarget          | Center of view pointed to by camera         | Values: \([x, y, z]\) axes coordinates  
Default: automatically determined by MATLAB |
| CameraTargetMode      | Uses MATLAB or user-specified camera target | Values: auto, manual  
Default: auto |
| CameraUpVector        | Direction that is oriented up               | Values: \([x, y, z]\) axes coordinates  
Default: automatically determined by MATLAB |
| CameraUpVectorMode    | Uses MATLAB or user-specified camera up vector | Values: auto, manual  
Default: auto |
| CameraViewAngle       | Camera field of view                       | Value: angle in degrees between 0 and 180  
Default: automatically determined by MATLAB |
| CameraViewAngleMode   | Uses MATLAB or user-specified camera view angle | Values: auto, manual  
Default: auto |
| Projection            | Selects type of projection                  | Values: orthographic, perspective  
Default: orthographic |

### Controlling the Axes Aspect Ratio
### Property Name | Property Description | Property Value
--- | --- | ---
DataAspectRatio | Relative scaling of data units | Values: three relative values \([dx \ dy \ dz]\)  
Default: automatically determined by MATLAB

DataAspectRatioMode | Uses MATLAB or user-specified data aspect ratio | Values: auto, manual  
Default: auto

PlotBoxAspectRatio | Relative scaling of axes plot box | Values: three relative values \([dx \ dy \ dz]\)  
Default: automatically determined by MATLAB

PlotBoxAspectRatioMode | Uses MATLAB or user-specified plot box aspect ratio | Values: auto, manual  
Default: auto

### Controlling Callback Routine Execution

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| BusyAction          | Specifies how to handle events that interrupt executing callback routines | Values: cancel, queue  
Default: queue

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| ButtonDownFcn       | Defines a callback routine that executes when a button is pressed over the axes | Values: string or function handle  
Default: an empty string

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| CreateFcn           | Defines a callback routine that executes when an axes is created          | Values: string or function handle  
Default: an empty string

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| DeleteFcn           | Defines a callback routine that executes when an axes is deleted          | Values: string or function handle  
Default: an empty string

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| Interruptible       | Controls whether an executing callback routine can be interrupted         | Values: on, off  
Default: on

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UICContextMenu</td>
<td>Associates a context menu with the axes</td>
<td>Values: handle of a Uicontextmenu</td>
</tr>
</tbody>
</table>
### Specifying the Rendering Mode

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| DrawMode      | Specifies the rendering method to use with the Painters renderer | Values: normal, fast  
Default: normal |

### Targeting Axes for Graphics Display

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| HandleVisibility| Controls access to a specific axes handle | Values: on, callback, off  
Default: on |
| NextPlot        | Determines the eligibility of the axes for displaying graphics | Values: add, replace, replacechildren  
Default: replace |

### Properties that Specify Transparency

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALim</td>
<td>Alpha axis limits</td>
<td>Values: [amin amax]</td>
</tr>
</tbody>
</table>
| ALimMode      | Alpha axis limits mode | Values: auto | manual  
Default: auto |

### Properties that Specify Color

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| AmbientLightColor        | Color of the background light in a scene | Values: ColorSpec  
Default: [1 1 1] |
| CLim                     | Controls how data is mapped to colormap | Values: [cmin cmax]  
Default: automatically determined by MATLAB |
| CLimMode                 | Uses MATLAB or user-specified values for CLim | Values: auto, manual  
Default: auto |
| Color                    | Color of the axes background          | Values: none, ColorSpec  
Default: none |
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Description</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| ColorOrder         | Line colors used for multiline plots                      | Value: m-by-3 matrix of RGB values  
Default: depends on color scheme used                                                                         |
| XColor, YColor, ZColor | Colors of the axis lines and tick marks                   | Values: ColorSpec  
Default: depends on current color scheme                                                                             |
Axes 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.

Axes Property Descriptions

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

**ActivePositionProperty**

{outerposition} | position

*Use OuterPosition or Position property for resize. ActivePositionProperty specifies which property MATLAB uses to determine the size of the axes when the figure is resized (interactively or during a printing or exporting operation).

See OuterPosition and Position for more information.

**ALim**

[amin, amax]

*Alpha axis limits.* A two-element vector that determines how MATLAB maps the AlphaData values of surface, patch, and image objects to the figure's alphamap. amin is the value of the data mapped to the first alpha value in the alphamap, and amax is the value of the data mapped to the last alpha value in the alphamap. Data values in between are linearly interpolated across the alphamap, while data values outside are clamped to either the first or last alphamap value, whichever is closest.

When **ALimMode** is auto (the default), MATLAB assigns amin the minimum data value and amax the maximum data value in the graphics object's AlphaData. This maps AlphaData elements with minimum data values to the first alphamap entry and those with maximum data values to the last alphamap entry. Data values in between are mapped linearly to the values.

If the axes contains multiple graphics objects, MATLAB sets **ALim** to span the range of all objects' AlphaData (or FaceVertexAlphaData for patch objects).

**ALimMode**

{auto} | manual

*Alpha axis limits mode.* In auto mode, MATLAB sets the **ALim** property to span the AlphaData limits of the graphics objects displayed in the axes. If **ALimMode**
is manual, MATLAB does not change the value of \texttt{ALim} when the \texttt{AlphaData} limits of axes children change. Setting the \texttt{ALim} property sets \texttt{ALimMode} to manual.

\textbf{AmbientLightColor} \hspace{0.5cm} ColorSpec

The background light in a scene. Ambient light is a directionless light that shines uniformly on all objects in the axes. However, if there are no visible light objects in the axes, MATLAB does not use \texttt{AmbientLightColor}. If there are light objects in the axes, the \texttt{AmbientLightColor} is added to the other light sources.

\textbf{AspectRatio} \hspace{0.5cm} (Obsolete)

This property produces a warning message when queried or changed. It has been superseded by the \texttt{DataAspectRatio}[\texttt{Mode}] and \texttt{PlotBoxAspectRatio}[\texttt{Mode}] properties.

\textbf{BeingDeleted} \hspace{0.5cm} on \hspace{0.5cm} \{off\}

This object is being deleted. The \texttt{BeingDeleted} property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB sets the \texttt{BeingDeleted} property to on when the object’s delete function callback is called (see the \texttt{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 if the objects are going to be deleted, and therefore, can check the object’s \texttt{BeingDeleted} property before acting.

\textbf{Box} \hspace{0.5cm} on \hspace{0.5cm} \{off\}

Axes box mode. This property specifies whether to enclose the axes extent in a box for 2-D views or a cube for 3-D views. The default is to not display the box.

\textbf{BusyAction} \hspace{0.5cm} cancel \hspace{0.5cm} \{queue\}

Callback routine interruption. The \texttt{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 \texttt{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 \texttt{Interruptible} property is off, the \texttt{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 within the axes, but not over another graphics object displayed in the axes. For 3-D views, the active area is defined by a rectangle that encloses the axes.

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.

**CameraPosition**  
[x, y, z] axes coordinates

*The location of the camera.* This property defines the position from which the camera views the scene. Specify the point in axes coordinates.

If you fix CameraViewAngle, you can zoom in and out on the scene by changing the CameraPosition, moving the camera closer to the CameraTarget to zoom in and farther away from the CameraTarget to zoom out. As you change the CameraPosition, the amount of perspective also changes, if Projection is perspective. You can also zoom by changing the CameraViewAngle; however, this does not change the amount of perspective in the scene.

**CameraPositionMode**  
{auto} | manual

*Auto or manual CameraPosition.* When set to auto, MATLAB automatically calculates the CameraPosition such that the camera lies a fixed distance from the CameraTarget along the azimuth and elevation specified by view. Setting a value for CameraPosition sets this property to manual.

**CameraTarget**  
[x, y, z] axes coordinates

*Camera aiming point.* This property specifies the location in the axes that the camera points to. The CameraTarget and the CameraPosition define the vector (the view axis) along which the camera looks.
**Axes Properties**

**CameraTargetMode**  
/auto} | manual

*Auto or manual CameraTarget placement.* When this property is auto, MATLAB automatically positions the CameraTarget at the centroid of the axes plot box. Specifying a value for CameraTarget sets this property to manual.

**CameraUpVector**  
[x, y, z] axes coordinates

*Camera rotation.* This property specifies the rotation of the camera around the viewing axis defined by the CameraTarget and the CameraPosition properties. Specify CameraUpVector as a three-element array containing the x, y, and z components of the vector. For example, [0 1 0] specifies the positive y-axis as the up direction.

The default CameraUpVector is [0 0 1], which defines the positive z-axis as the up direction.

**CameraUpVectorMode**  
/auto} | manual

*Default or user-specified up vector.* When CameraUpVectorMode is auto, MATLAB uses a value of [0 0 1] (positive z-direction is up) for 3-D views and [0 1 0] (positive y-direction is up) for 2-D views. Setting a value for CameraUpVector sets this property to manual.

**CameraViewAngle**  
scalar greater than 0 and less than or equal to 180 (angle in degrees)

*The field of view.* This property determines the camera field of view. Changing this value affects the size of graphics objects displayed in the axes, but does not affect the degree of perspective distortion. The greater the angle, the larger the field of view, and the smaller objects appear in the scene.

**CameraViewAngleMode**  
/auto} | manual

*Auto or manual CameraViewAngle.* When in auto mode, MATLAB sets CameraViewAngle to the minimum angle that captures the entire scene (up to 180°).
Axes Properties

The following table summarizes MATLAB automatic camera behavior.

<table>
<thead>
<tr>
<th>CameraView Angle</th>
<th>Camera Target</th>
<th>Camera Position</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>auto</td>
<td>auto</td>
<td>CameraTarget is set to plot box centroid, CameraViewAngle is set to capture entire scene, CameraPosition is set along the view axis.</td>
</tr>
<tr>
<td>auto</td>
<td>auto</td>
<td>manual</td>
<td>CameraTarget is set to plot box centroid, CameraViewAngle is set to capture entire scene.</td>
</tr>
<tr>
<td>auto</td>
<td>manual</td>
<td>auto</td>
<td>CameraViewAngle is set to capture entire scene, CameraPosition is set along the view axis.</td>
</tr>
<tr>
<td>auto</td>
<td>manual</td>
<td>manual</td>
<td>CameraViewAngle is set to capture entire scene.</td>
</tr>
<tr>
<td>manual</td>
<td>auto</td>
<td>auto</td>
<td>CameraTarget is set to plot box centroid, CameraPosition is set along the view axis.</td>
</tr>
<tr>
<td>manual</td>
<td>auto</td>
<td>manual</td>
<td>CameraTarget is set to plot box centroid.</td>
</tr>
<tr>
<td>manual</td>
<td>manual</td>
<td>auto</td>
<td>CameraPosition is set along the view axis.</td>
</tr>
<tr>
<td>manual</td>
<td>manual</td>
<td>manual</td>
<td>All camera properties are user-specified.</td>
</tr>
</tbody>
</table>

Children

*Children of the axes.* A vector containing the handles of all graphics objects rendered within the axes (whether visible or not). The graphics objects that can be children of axes are images, lights, lines, patches, rectangles, surfaces, and text. You can change the order of the handles and thereby change the stacking of the objects on the display.

The text objects used to label the x-, y-, and z-axes are also children of axes, but their HandleVisibility properties are set to callback. This means their handles do not show up in the axes Children property unless you set the Root ShowHiddenHandles property to on.

When an object’s HandleVisibility property is set to off, it is not listed in its parent’s Children property. See HandleVisibility for more information.
Axes Properties

**Clim**  
\([\text{cmin}, \text{cmax}]\)

*Color axis limits.* A two-element vector that determines how MATLAB maps the \(\text{CData}\) values of surface and patch objects to the figure's colormap. \(\text{cmin}\) is the value of the data mapped to the first color in the colormap, and \(\text{cmax}\) is the value of the data mapped to the last color in the colormap. Data values in between are linearly interpolated across the colormap, while data values outside are clamped to either the first or last colormap color, whichever is closest.

When \(\text{CLimMode}\) is \(\text{auto}\) (the default), MATLAB assigns \(\text{cmin}\) the minimum data value and \(\text{cmax}\) the maximum data value in the graphics object's \(\text{CData}\). This maps \(\text{CData}\) elements with minimum data value to the first colormap entry and with maximum data value to the last colormap entry.

If the axes contains multiple graphics objects, MATLAB sets \(\text{CLim}\) to span the range of all objects' \(\text{CData}\).

**CLimMode**  
\{\text{auto} \mid \text{manual}\}

*Color axis limits mode.* In \(\text{auto}\) mode, MATLAB sets the \(\text{CLim}\) property to span the \(\text{CData}\) limits of the graphics objects displayed in the axes. If \(\text{CLimMode}\) is \(\text{manual}\), MATLAB does not change the value of \(\text{CLim}\) when the \(\text{CData}\) limits of axes children change. Setting the \(\text{CLim}\) property sets this property to \(\text{manual}\).

**Clipping**  
\{\text{on} \mid \text{off}\}

This property has no effect on axes.

**Color**  
\{\text{none} \mid \text{ColorSpec}\}

*Color of the axes back planes.* Setting this property to \text{none} means the axes is transparent and the figure color shows through. A \text{ColorSpec} is a three-element RGB vector or one of the MATLAB predefined names. Note that while the default value is \text{none}, the \text{matlabrc.m} file may set the axes color to a specific color.

**ColorOrder**  
\(m\times3\) matrix of RGB values

*Colors to use for multiline plots.* \text{ColorOrder} is an \(m\times3\) matrix of RGB values that define the colors used by the \text{plot} and \text{plot3} functions to color each line plotted. If you do not specify a line color with \text{plot} and \text{plot3}, these functions cycle through the \text{ColorOrder} to obtain the color for each line plotted. To obtain the current \text{ColorOrder}, which may be set during startup, get the property value:
Axes Properties

get(gca,'ColorOrder')

Note that if the axes NextPlot property is set to replace (the default), high-level functions like plot reset the ColorOrder property before determining the colors to use. If you want MATLAB to use a ColorOrder that is different from the default, set NextPlot to replacechildren. You can also specify your own default ColorOrder.

CreateFcn string or function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates an axes object. You must define this property as a default value for axes. For example, the statement

```matlab
set(0, 'DefaultAxesCreateFcn', 'set(gca, ''Color'', ''b'')')
```

defines a default value on the Root level that sets the current axes background color to blue whenever you (or MATLAB) create an axes. MATLAB executes this routine after setting all properties for the axes. Setting this property on an existing axes object has no effect.

The handle of the object whose CreateFcn 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.

CurrentPoint 2-by-3 matrix

Location of last button click, in axes data units. A 2-by-3 matrix containing the coordinates of two points defined by the location of the pointer. These two points lie on the line that is perpendicular to the plane of the screen and passes through the pointer. The 3-D coordinates are the points, in the axes coordinate system, where this line intersects the front and back surfaces of the axes volume (which is defined by the axes x, y, and z limits).

The returned matrix is of the form

```
[xback yback zback]
[xfront yfront zfront]
```

MATLAB updates the CurrentPoint property whenever a button-click event occurs. The pointer does not have to be within the axes, or even the figure.
Axes Properties

window; MATLAB returns the coordinates with respect to the requested axes regardless of the pointer location.

DataAspectRatio [dx dy dz]

*Relative scaling of data units.* A three-element vector controlling the relative scaling of data units in the $x$, $y$, and $z$ directions. For example, setting this property to $[1 \ 2 \ 1]$ causes the length of one unit of data in the $x$ direction to be the same length as two units of data in the $y$ direction and one unit of data in the $z$ direction.

Note that the DataAspectRatio property interacts with the PlotBoxAspectRatio, XLimMode, YLimMode, and ZLimMode properties to control how MATLAB scales the $x$-, $y$-, and $z$-axis. Setting the DataAspectRatio will disable the stretch-to-fill behavior if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all auto. The following table describes the interaction between properties when stretch-to-fill behavior is disabled.

<table>
<thead>
<tr>
<th>X-, Y-, Z-Limits</th>
<th>DataAspectRatio</th>
<th>PlotBoxAspectRatio</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>auto</td>
<td>auto</td>
<td>Limits chosen to span data range in all dimensions.</td>
</tr>
<tr>
<td>auto</td>
<td>auto</td>
<td>manual</td>
<td>Limits chosen to span data range in all dimensions. DataAspectRatio is modified to achieve the requested PlotBoxAspectRatio within the limits selected by MATLAB.</td>
</tr>
<tr>
<td>auto</td>
<td>manual</td>
<td>auto</td>
<td>Limits chosen to span data range in all dimensions. PlotBoxAspectRatio is modified to achieve the requested DataAspectRatio within the limits selected by MATLAB.</td>
</tr>
<tr>
<td>auto</td>
<td>manual</td>
<td>manual</td>
<td>Limits chosen to completely fit and center the plot within the requested PlotBoxAspectRatio given the requested DataAspectRatio (this may produce empty space around 2 of the 3 dimensions).</td>
</tr>
</tbody>
</table>
### Axes Properties

<table>
<thead>
<tr>
<th>X-, Y-, Z-Limits</th>
<th>DataAspectRatio</th>
<th>PlotBoxAspectRatio</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>manual</td>
<td>auto</td>
<td>auto</td>
<td>Limits are honored. The DataAspectRatio and PlotBoxAspectRatio are modified as necessary.</td>
</tr>
<tr>
<td>manual</td>
<td>auto</td>
<td>manual</td>
<td>Limits and PlotBoxAspectRatio are honored. The DataAspectRatio is modified as necessary.</td>
</tr>
<tr>
<td>manual</td>
<td>manual</td>
<td>auto</td>
<td>Limits and DataAspectRatio are honored. The PlotBoxAspectRatio is modified as necessary. The 2 automatic limits are selected to honor the specified aspect ratios and limit. See “Examples.”</td>
</tr>
<tr>
<td>1 manual 2 auto</td>
<td>manual</td>
<td>manual</td>
<td>Limits and DataAspectRatio are honored; the PlotBoxAspectRatio is ignored.</td>
</tr>
</tbody>
</table>

**DataAspectRatioMode**  
{auto} | manual  
*User or MATLAB controlled data scaling.* This property controls whether the values of the DataAspectRatio property are user defined or selected automatically by MATLAB. Setting values for the DataAspectRatio property automatically sets this property to `manual`. Changing DataAspectRatioMode to `manual` disables the stretch-to-fill behavior if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all `auto`.

**DeleteFcn**  
string or function handle  
*Delete axes callback routine.* A callback routine that executes when the axes object is deleted (e.g., when you issue a delete command). MATLAB executes the routine 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.
Axes Properties

**DrawMode**

*{normal} | fast*

*Rendering method.* This property controls the method MATLAB uses to render graphics objects displayed in the axes, when the figure **Renderer** property is **painters**.

- *normal* mode draws objects in back to front ordering based on the current view in order to handle hidden surface elimination and object intersections.
- *fast* mode draws objects in the order in which you specify the drawing commands, without considering the relationships of the objects in three dimensions. This results in faster rendering because it requires no sorting of objects according to location in the view, but may produce undesirable results because it bypasses the hidden surface elimination and object intersection handling provided by normal **DrawMode**.

When the figure **Renderer** is **zbuffer**, **DrawMode** is ignored, and hidden surface elimination and object intersection handling are always provided.

**FontAngle**

*{normal} | italic | oblique*

*Select italic or normal font.* This property selects the character slant for axes text. *normal* specifies a nonitalic font. *italic* and *oblique* specify italic font.

**FontName**

*A name such as Courier or the string FixedWidth*

*Font family name.* The font family name specifying the font to use for axes labels. To display and print properly, **FontName** must be a font that your system supports. Note that the x-, y-, and z-axis labels are not displayed in a new font until you manually reset them (by setting the **XLabel**, **YLabel**, and **ZLabel** properties or by using the **xlabel**, **ylabel**, or **zlabel** command). Tick mark labels change immediately.

**Specifying a Fixed-Width Font**

If you want an axes to use a fixed-width font that looks good in any locale, you should set **FontName** to the string **FixedWidth**:

```matlab
set(axes_handle,'FontName','FixedWidth')
```

This eliminates the need to hardcode 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**
Axes Properties

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    Font size specified in FontUnits

Font size. An integer specifying the font size to use for axes labels and titles, in
units determined by the FontUnits property. The default point size is 12. The
x-, y-, and z-axis text labels are not displayed in a new font size until you
manually reset them (by setting the XLabel, YLabel, or ZLabel properties or by
using the xlabel, ylabel, or zlabel command). Tick mark labels change
immediately.

FontUnits   {points} | normalized | inches |
            | centimeters | pixels

Units used to interpret the FontSize property. When set to normalized,
MATLAB interprets the value of FontSize as a fraction of the height of the
axes. For example, a normalized FontSize of 0.1 sets the text characters to a
font whose height is one tenth of the axes’ height. The default units (points),
are equal to 1/72 of an inch.

FontWeight  {normal} | bold | light | demi

Select bold or normal font. The character weight for axes text. The x-, y-, and
z-axis text labels are not displayed in bold until you manually reset them (by
setting the XLabel, YLabel, and ZLabel properties or by using the xlabel,
ylabel, or zlabel commands). Tick mark labels change immediately.

GridLineStyle  − | −− | {−} | −−− | none

Line style used to draw grid lines. The line style is a string consisting of a
character, in quotes, specifying solid lines (−), dashed lines (−−), dotted lines(−),
or dash-dot lines (−−−). The default grid line style is dotted. To turn on grid lines,
use the grid command.

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
Axes Properties

HandleVisibility is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when HandleVisibility is on.

Setting HandleVisibility to callback causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

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

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

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

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

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

**HitTest**

{on} | off

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

**Interruptible**  
{on} | off

*Callback routine interruption mode.* The `Interruptible` property controls whether an axes 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.

Setting `Interruptible` to on allows any graphics object’s callback routine to interrupt callback routines originating from an axes 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.

**Layer**  
{bottom} | top

*Draw axis lines below or above graphics objects.* This property determines if axis lines and tick marks are drawn on top or below axes children objects for any 2-D view (i.e., when you are looking along the x-, y-, or z-axis). This is useful for placing grid lines and tick marks on top of images.

**LineStyleOrder**  
LineSpec (default: a solid line ‘-‘)

*Order of line styles and markers used in a plot.* This property specifies which line styles and markers to use and in what order when creating multiple-line plots. For example,

```matlab
set(gca,'LineStyleOrder', '-*|--|')
```

sets `LineStyleOrder` to solid line with asterisk marker, dotted line, and hollow circle marker. The default is (–), which specifies a solid line for all data plotted. Alternatively, you can create a cell array of character strings to define the line styles:

```matlab
set(gca,'LineStyleOrder',{'-*','--','o'})
```

MATLAB supports four line styles, which you can specify any number of times in any order. MATLAB cycles through the line styles only after using all colors defined by the `ColorOrder` property. For example, the first eight lines plotted use the different colors defined by `ColorOrder` with the first line style. MATLAB then cycles through the colors again, using the second line style specified, and so on.
Axes Properties

You can also specify line style and color directly with the plot and plot3 functions or by altering the properties of the line or lineseries objects after creating the graph.

High-Level Functions andLineStyleOrder
Note that, if the axes NextPlot property is set to replace (the default), high-level functions like plot reset theLineStyleOrder property before determining the line style to use. If you want MATLAB to use aLineStyleOrder that is different from the default, set NextPlot to replacechildren.

Specifying a DefaultLineStyleOrder
You can also specify your own defaultLineStyleOrder. For example, this statement

```matlab
set(0,'DefaultAxesLineStyleOrder',{'-*',':','o'})
```

creates a default value for the axesLineStyleOrder that is not reset by high-level plotting functions.

LineWidth
line width in points

Width of axis lines. This property specifies the width, in points, of the x-, y-, and z-axis lines. The default line width is 0.5 points (1 point = 1/72 inch).

MinorGridLineStyle – | − − | {·} | − . | none

Line style used to draw minor grid lines. The line style is a string consisting of one or more characters, in quotes, specifying solid lines (–), dashed lines (—), dotted lines (:), or dash-dot lines (–·). The default minor grid line style is dotted. To turn on minor grid lines, use the grid minor command.

NextPlot
add | {replace} | replacechildren

Where to draw the next plot. This property determines how high-level plotting functions draw into an existing axes.

• add — Use the existing axes to draw graphics objects.
• replace — Reset all axes properties except Position to their defaults and delete all axes children before displaying graphics (equivalent to cla reset).
• replacechildren — Remove all child objects, but do not reset axes properties (equivalent to cla).
Axes Properties

The `newplot` function simplifies the use of the `NextPlot` property and is used by M-file functions that draw graphs using only low-level object creation routines. See the M-file `pcolor.m` for an example. Note that figure graphics objects also have a `NextPlot` property.

**OuterPosition** four-element vector

*Position of axes including labels, title, and a margin.* A four-element vector specifying a rectangle that locates the outer bounds of the axes, including axis labels, the title, and a margin. The vector is defined as follows:

\[ [\text{left} \ \text{bottom} \ \text{width} \ \text{height}] \]

where `left` and `bottom` define the distance from the lower-left corner of the figure window to the lower-left corner of the rectangle. `width` and `height` are the dimensions of the rectangle.

The following picture shows the region defined by the `OuterPosition` enclosed in a yellow rectangle.

The yellow rectangle shows the extent of the `OuterPosition`.

The green rectangle shows the extent of the `Position`.

![Plot of a simple mathematical function](image)

\[ y = x^2 \]

\[ x = -\pi \text{ to } 2\pi \]
Axes Properties

When `ActivePositionProperty` is set to `OuterPosition` (the default), none of the text is clipped when you resize the figure. The default value of `[0 0 1 1]` (normalized units) includes the interior of the figure.

All measurements are in units specified by the `Units` property.

See the `TightInset` property for related information.

See Automatic Axes Resize for more information.

**Parent**

`Axes parent`. The handle of the axes’ parent object. The parent of an axes object is the figure in which it is displayed or the uipanel object that contains it. The utility function `gcf` returns the handle of the current axes `Parent`. You can reparent axes to other figure or uipanel objects.

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

**PlotBoxAspectRatio**

`[px py pz]`

*Relative scaling of axes plot box.* A three-element vector controlling the relative scaling of the plot box in the x, y, and z directions. The plot box is a box enclosing the axes data region as defined by the x-, y-, and z-axis limits.

Note that the `PlotBoxAspectRatio` property interacts with the `DataAspectRatio`, `XLimMode`, `YLimMode`, and `ZLimMode` properties to control the way graphics objects are displayed in the axes. Setting the `PlotBoxAspectRatio` property disables stretch-to-fill behavior, if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`.

**PlotBoxAspectRatioMode**

`{auto} | manual`

*User or MATLAB controlled axis scaling.* This property controls whether the values of the `PlotBoxAspectRatio` property are user defined or selected automatically by MATLAB. Setting values for the `PlotBoxAspectRatio` property automatically sets this property to `manual`. Changing the `PlotBoxAspectRatioMode` to `manual` disables stretch-to-fill behavior if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`. 
Axes Properties

**Position**

four-element vector

*Position of axes.* A four-element vector specifying a rectangle that locates the axes within the figure window. The vector is of the form

\[
\text{[left bottom width height]}
\]

where `left` and `bottom` define the distance from the lower-left corner of the figure window to the lower-left corner of the rectangle. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the Units property.

When axes stretch-to-fill behavior is enabled (when DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all auto), the axes are stretched to fill the Position rectangle. When stretch-to-fill is disabled, the axes are made as large as possible, while obeying all other properties, without extending outside the Position rectangle.

See the OuterPosition property for related information.

**Projection**

{orthographic} | perspective

*Type of projection.* This property selects between two projection types:

- **orthographic** — This projection maintains the correct relative dimensions of graphics objects with regard to the distance a given point is from the viewer. Parallel lines in the data are drawn parallel on the screen.

- **perspective** — This projection incorporates foreshortening, which allows you to perceive depth in 2-D representations of 3-D objects. Perspective projection does not preserve the relative dimensions of objects; a distant line segment is displayed smaller than a nearer line segment of the same length. Parallel lines in the data may not appear parallel on screen.

**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 axes has been selected.

**SelectionHighlight**

{on} | off

*Objects are highlighted when selected.* When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four
corner handles. When `SelectionHighlight` is off, MATLAB does not draw the handles.

**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 want to direct all graphics output from an M-file to a particular axes, regardless of user actions that may have changed the current axes. To do this, identify the axes with a Tag:

```matlab
axes('Tag','Special Axes')
```

Then make that axes the current axes before drawing by searching for the Tag with `findobj`:

```matlab
axes(findobj('Tag','Special Axes'))
```

**TickDir**

in | out

*Direction of tick marks.* For 2-D views, the default is to direct tick marks inward from the axis lines; 3-D views direct tick marks outward from the axis line.

**TickDirMode**

{auto} | manual

*Automatic tick direction control.* In auto mode, MATLAB directs tick marks inward for 2-D views and outward for 3-D views. When you specify a setting for `TickDir`, MATLAB sets `TickDirMode` to manual. In manual mode, MATLAB does not change the specified tick direction.

**TickLength**

[2DLength 3DLength]

*Length of tick marks.* A two-element vector specifying the length of axes tick marks. The first element is the length of tick marks used for 2-D views and the second element is the length of tick marks used for 3-D views. Specify tick mark lengths in units normalized relative to the longest of the visible X-, Y-, or Z-axis annotation lines.
Axes Properties

TightInset [left bottom right top] Read only

Margins added to Position to include text labels. The values of this property are the distances between the bounds of the Position property and the extent of the axes text labels and title. When added to the Position width and height values, the TightInset defines the tightest bounding box that encloses the axes and its labels and title.

See Automatic Axes Resize for more information.

Title handle of text object

Axes title. The handle of the text object that is used for the axes title. You can use this handle to change the properties of the title text or you can set Title to the handle of an existing text object. For example, the following statement changes the color of the current title to red:

```matlab
set(get(gca,'Title'),'Color','r')
```

To create a new title, set this property to the handle of the text object you want to use:

```matlab
set(gca,'Title',text('String','New Title','Color','r'))
```

However, it is generally simpler to use the title command to create or replace an axes title:

```matlab
title('New Title','Color','r') % Make text color red
title({'This title','has 2 lines'}) % Two line title
```

Type string (read only)

Type of graphics object. This property contains a string that identifies the class of graphics object. For axes objects, Type is always set to 'axes'.

UIContextMenu handle of a uicontextmenu object

Associate a context menu with the axes. Assign this property the handle of a Uicontextmenu object created in the axes’ parent figure. Use the uicontextmenu function to create the context menu. MATLAB displays the context menu whenever you right-click over the axes.

Units inches | centimeters | {normalized} | points | pixels | characters

Position units. The units used to interpret the Position property. All units are measured from the lower left corner of the figure window.
Axes Properties

- normalized units map the lower left corner of the figure window 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).
- Character units are defined by characters from the default system font; the width of one character is the width of the letter x, and the height of one character is the distance between the baselines of two lines of text.

UserData matrix
User-specified data. This property can be any data you want to associate with the axes object. The axes does not use this property, but you can access it using the set and get functions.

View Obsolete
The functionality provided by the View property is now controlled by the axes camera properties — CameraPosition, CameraTarget, CameraUpVector, and CameraViewAngle. See the view command.

Visible {on} | off
Visibility of axes. By default, axes are visible. Setting this property to off prevents axis lines, tick marks, and labels from being displayed. The Visible property does not affect children of axes.

XAxisLocation top | {bottom}
Location of x-axis tick marks and labels. This property controls where MATLAB displays the x-axis tick marks and labels. Setting this property to top moves the x-axis to the top of the plot from its default position at the bottom.

YAxisLocation right | {left}
Location of y-axis tick marks and labels. This property controls where MATLAB displays the y-axis tick marks and labels. Setting this property to right moves the y-axis to the right side of the plot from its default position on the left side. See the plotyy function for a simple way to use two y-axes.

Properties That Control the X-, Y-, or Z-Axis

XColor, YColor, ZColor ColorSpec
Color of axis lines. A three-element vector specifying an RGB triple, or a predefined MATLAB color string. This property determines the color of the axis
Axes Properties

lines, tick marks, tick mark labels, and the axis grid lines of the respective x-, y-, and z-axis. The default color axis color is black. See ColorSpec for details on specifying colors.

**XDir, YDir, ZDir**  
\{normal\} | reverse  
*Direction of increasing values.* A mode controlling the direction of increasing axis values. Axes form a right-hand coordinate system. By default,

- x-axis values increase from left to right. To reverse the direction of increasing x values, set this property to reverse.
  
  ```set(gca,'XDir','reverse')```

- y-axis values increase from bottom to top (2-D view) or front to back (3-D view). To reverse the direction of increasing y values, set this property to reverse.
  
  ```set(gca,'YDir','reverse')```

- z-axis values increase pointing out of the screen (2-D view) or from bottom to top (3-D view). To reverse the direction of increasing z values, set this property to reverse.
  
  ```set(gca,'ZDir','reverse')```

**XGrid, YGrid, ZGrid**  
on | \{off\}  
*Axis gridline mode.* When you set any of these properties to on, MATLAB draws grid lines perpendicular to the respective axis (i.e., along lines of constant x, y, or z values). Use the grid command to set all three properties on or off at once.

  ```set(gca,'XGrid','on')```

**XLabel, YLabel, ZLabel**  
handle of text object  
*Axis labels.* The handle of the text object used to label the x-, y-, or z-axis, respectively. To assign values to any of these properties, you must obtain the handle to the text string you want to use as a label. This statement defines a text object and assigns its handle to the XLabel property:

  ```set(get(gca,'XLabel'),'String','axis label')```

MATLAB places the string ‘axis label’ appropriately for an x-axis label. Any text object whose handle you specify as an XLabel, YLabel, or ZLabel property is moved to the appropriate location for the respective label.
Axes Properties

Alternatively, you can use the xlabel, ylabel, and zlabel functions, which generally provide a simpler means to label axis lines.

**XLimit, YLimit, ZLimit**  
[minumum maximum]  
*Axis limits.* A two-element vector specifying the minimum and maximum values of the respective axis.

Changing these properties affects the scale of the x-, y-, or z-dimension as well as the placement of labels and tick marks on the axis. The default values for these properties are [0 1].

**XLimitMode, YLimitMode, ZLimitMode**  
{auto} | manual  
*MATLAB or user-controlled limits.* The axis limits mode determines whether MATLAB calculates axis limits based on the data plotted (i.e., the XData, YData, or ZData of the axes children) or uses the values explicitly set with the XLimit, YLimit, or ZLimit property, in which case, the respective limits mode is set to manual.

**XMinorGrid, YMinorGrid, ZMinorGrid**  
on | {off}  
*Enable or disable minor gridlines.* When set to on, MATLAB draws gridlines aligned with the minor tick marks of the respective axis. Note that you do not have to enable minor ticks to display minor grids.

**XMinorTick, YMinorTick, ZMinorTick**  
on | {off}  
*Enable or disable minor tick marks.* When set to on, MATLAB draws tick marks between the major tick marks of the respective axis. MATLAB automatically determines the number of minor ticks based on the space between the major ticks.

**XScale, YScale, ZScale**  
{linear} | log  
*Axis scaling.* Linear or logarithmic scaling for the respective axis. See also loglog, semilogx, and semilogy.

**XTick, YTick, ZTick**  
vector of data values locating tick marks  
*Tick spacing.* A vector of x-, y-, or z-data values that determine the location of tick marks along the respective axis. If you do not want tick marks displayed, set the respective property to the empty vector, []. These vectors must contain monotonically increasing values.
**Axes Properties**

**XTickLabel, YTickLabel, ZTickLabel**  
*string*

*Tick labels.* A matrix of strings to use as labels for tick marks along the respective axis. These labels replace the numeric labels generated by MATLAB. If you do not specify enough text labels for all the tick marks, MATLAB uses all of the labels specified, then reuses the specified labels.

For example, the statement

```matlab
set(gca,'XTickLabel', {'One'; 'Two'; 'Three'; 'Four'})
```

labels the first four tick marks on the x-axis and then reuses the labels until all ticks are labeled.

Labels can be specified as cell arrays of strings, padded string matrices, string vectors separated by vertical slash characters, or as numeric vectors (where each number is implicitly converted to the equivalent string using `num2str`). All of the following are equivalent:

```matlab
set(gca,'XTickLabel', {'1'; '10'; '100'})
set(gca,'XTickLabel','1|10|100')
set(gca,'XTickLabel',[1;10;100])
set(gca,'XTickLabel', ['1 ';'10 ';'100'])
```

Note that tick labels do not interpret TeX character sequences (however, the Title, XLabel, YLabel, and ZLabel properties do).

**XTickMode, YTickMode, ZTickMode**  
*manual* | {auto}

*MATLAB or user-controlled tick spacing.* The axis tick modes determine whether MATLAB calculates the tick mark spacing based on the range of data for the respective axis (auto mode) or uses the values explicitly set for any of the XTick, YTick, and ZTick properties (manual mode). Setting values for the XTick, YTick, or ZTick properties sets the respective axis tick mode to manual.

**XTickLabelMode, YTickLabelMode, ZTickLabelMode**  
*manual*

*MATLAB or user-determined tick labels.* The axis tick mark labeling mode determines whether MATLAB uses numeric tick mark labels that span the range of the plotted data (auto mode) or uses the tick mark labels specified with the XTickLabel, YTickLabel, or ZTickLabel property (manual mode). Setting values for the XTickLabel, YTickLabel, or ZTickLabel property sets the respective axis tick label mode to manual.
**Purpose**

Axis scaling and appearance

**Syntax**

```matlab
axis([xmin xmax ymin ymax])
axis([xmin xmax ymin ymax zmin zmax cmin cmax])
```

```matlab
v = axis
```

```matlab
axis auto
axis manual
axis tight
axis fill
```

```matlab
axis ij
axis xy
```

```matlab
axis equal
axis image
axis square
axis vis3d
axis normal
```

```matlab
axis off
axis on
axis(axes_handles,...)
```

```
[mode,visibility,direction] = axis('state')
```

**Description**

axis manipulates commonly used axes properties. (See Algorithm section.)

axis([xmin xmax ymin ymax]) sets the limits for the x- and y-axis of the current axes.

axis([xmin xmax ymin ymax zmin zmax cmin cmax]) sets the x-, y-, and z-axis limits and the color scaling limits (see caxis) of the current axes.

v = axis returns a row vector containing scaling factors for the x-, y-, and z-axis. v has four or six components depending on whether the current axes is 2-D or 3-D, respectively. The returned values are the current axes XLim, Ylim, and ZLim properties.
axis auto sets MATLAB to its default behavior of computing the current axes limits automatically, based on the minimum and maximum values of x, y, and z data. You can restrict this automatic behavior to a specific axis. For example, axis 'auto x' computes only the x-axis limits automatically; axis 'auto yz' computes the y- and z-axis limits automatically.

axis manual and axis(axis) freezes the scaling at the current limits, so that if hold is on, subsequent plots use the same limits. This sets the XLimMode, YLimMode, and ZLimMode properties to manual.

axis tight sets the axis limits to the range of the data.

axis fill sets the axis limits and PlotBoxAspectRatio so that the axes fill the position rectangle. This option has an effect only if PlotBoxAspectRatioMode or DataAspectRatioMode is manual.

axis ij places the coordinate system origin in the upper left corner. The i-axis is vertical, with values increasing from top to bottom. The j-axis is horizontal with values increasing from left to right.

axis xy draws the graph in the default Cartesian axes format with the coordinate system origin in the lower left corner. The x-axis is horizontal with values increasing from left to right. The y-axis is vertical with values increasing from bottom to top.

axis equal sets the aspect ratio so that the data units are the same in every direction. The aspect ratio of the x-, y-, and z-axis is adjusted automatically according to the range of data units in the x, y, and z directions.

axis image is the same as axis equal except that the plot box fits tightly around the data.

axis square makes the current axes region square (or cubed when three-dimensional). MATLAB adjusts the x-axis, y-axis, and z-axis so that they have equal lengths and adjusts the increments between data units accordingly.

axis vis3d freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretch-to-fill.
axis normal automatically adjusts the aspect ratio of the axes and the relative scaling of the data units so that the plot fits the figure's shape as well as possible.

axis off turns off all axis lines, tick marks, and labels.

axis on turns on all axis lines, tick marks, and labels.

axis(axes_handles,...) applies the axis command to the specified axes. For example, the following statements

```
    h1 = subplot(221);  
h2 = subplot(222);  
axis([h1 h2],'square')
```

set both axes to square.

```
[mode,visibility,direction] = axis('state')
```

returns three strings indicating the current setting of axes properties:

<table>
<thead>
<tr>
<th>Output Argument</th>
<th>Strings Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>'auto'</td>
</tr>
<tr>
<td>visibility</td>
<td>'on'</td>
</tr>
<tr>
<td>direction</td>
<td>'xy'</td>
</tr>
</tbody>
</table>

mode is auto if XLimMode, YLimMode, and ZLimMode are all set to auto. If XLimMode, YLimMode, or ZLimMode is manual, mode is manual.

**Examples**

The statements

```
    x = 0:.025:pi/2;  
    plot(x,tan(x),'-ro')
```

use the automatic scaling of the y-axis based on $y_{max} = \tan(1.57)$, which is well over 1000:
The right figure shows a more satisfactory plot after typing

```
axis([0 pi/2 0 5])
```

When you specify minimum and maximum values for the $x$-, $y$-, and $z$-axes, `axis` sets the `XLim`, `Ylim`, and `ZLim` properties for the current axes to the respective minimum and maximum values in the argument list. Additionally, the `XLimMode`, `YLimMode`, and `ZLimMode` properties for the current axes are set to `manual`.

`axis auto` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to `'auto'`.

`axis manual` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to `'manual'`. 
The following table shows the values of the axes properties set by `axis equal`, `axis normal`, `axis square`, and `axis image`.

<table>
<thead>
<tr>
<th>Axes Property</th>
<th>axis equal</th>
<th>axis normal</th>
<th>axis square</th>
<th>axis tightequal</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataAspectRatio</td>
<td>[1 1 1]</td>
<td>not set</td>
<td>not set</td>
<td>[1 1 1]</td>
</tr>
<tr>
<td>DataAspectRatioMode</td>
<td>manual</td>
<td>auto</td>
<td>auto</td>
<td>manual</td>
</tr>
<tr>
<td>PlotBoxAspectRatio</td>
<td>[3 4 4]</td>
<td>not set</td>
<td>[1 1 1]</td>
<td>auto</td>
</tr>
<tr>
<td>PlotBoxAspectRatioMode</td>
<td>manual</td>
<td>auto</td>
<td>manual</td>
<td>auto</td>
</tr>
<tr>
<td>Stretch-to-fill</td>
<td>disabled</td>
<td>active</td>
<td>disabled</td>
<td>disabled</td>
</tr>
</tbody>
</table>

**See Also**
axes, grid, subplot, xlim, ylim, zlim

Properties of axes graphics objects

“Axes Operations” for related functions
Purpose
Diagonal scaling to improve eigenvalue accuracy

Syntax

\[
[T,B] = \text{balance}(A)
\]

\[
[S,P,B] = \text{balance}(A)
\]

\[
B = \text{balance}(A)
\]

\[
B = \text{balance}(A,\text{'noperm'})
\]

Description

\[T,B] = \text{balance}(A)\] returns a similarity transformation \(T\) such that \(B = T\backslash A\backslash T\), and \(B\) has, as nearly as possible, approximately equal row and column norms. \(T\) is a permutation of a diagonal matrix whose elements are integer powers of two to prevent the introduction of round-off error. If \(A\) is symmetric, then \(B = A\) and \(T\) is the identity matrix.

\[S,P,B] = \text{balance}(A)\] returns the scaling vector \(S\) and the permutation vector \(P\) separately. The transformation \(T\) and balanced matrix \(B\) are obtained from \(A\), \(S\), and \(P\) by \(T(:,P) = \text{diag}(S)\) and \(B(P,P) = \text{diag}(1./S)*A*\text{diag}(S)\).

\[B = \text{balance}(A)\] returns just the balanced matrix \(B\).

\[B = \text{balance}(A, \text{'noperm'})\] scales \(A\) without permuting its rows and columns.

Remarks

Nonsymmetric matrices can have poorly conditioned eigenvalues. Small perturbations in the matrix, such as roundoff errors, can lead to large perturbations in the eigenvalues. The condition number of the eigenvector matrix,

\[
\text{cond}(V) = \text{norm}(V)\times\text{norm}(\text{inv}(V))
\]

where

\[ [V,T] = \text{eig}(A) \]

relates the size of the matrix perturbation to the size of the eigenvalue perturbation. Note that the condition number of \(A\) itself is irrelevant to the eigenvalue problem.

Balancing is an attempt to concentrate any ill conditioning of the eigenvector matrix into a diagonal scaling. Balancing usually cannot turn a nonsymmetric matrix into a symmetric matrix; it only attempts to make the norm of each row equal to the norm of the corresponding column.
Note  The MATLAB eigenvalue function, \( \text{eig}(A) \), automatically balances \( A \) before computing its eigenvalues. Turn off the balancing with \( \text{eig}(A, 'nobalance') \).

Examples

This example shows the basic idea. The matrix \( A \) has large elements in the upper right and small elements in the lower left. It is far from being symmetric.

\[
A = \begin{bmatrix}
1 & 100 & 10000; & .01 & 1 & 100; & .0001 & .01 & 1
\end{bmatrix}
\]

Balancing produces a diagonal matrix \( T \) with elements that are powers of two and a balanced matrix \( B \) that is closer to symmetric than \( A \).

\[
[T,B] = \text{balance}(A)
\]

To see the effect on eigenvectors, first compute the eigenvectors of \( A \), shown here as the columns of \( V \).

\[
[V,E] = \text{eig}(A); \quad V
\]
balance

Note that all three vectors have the first component the largest. This indicates \( V \) is badly conditioned; in fact \( \text{cond}(V) = 8.7766 \times 10^3 \). Next, look at the eigenvectors of \( B \).

\[
[V, E] = \text{eig}(B); V
\]

\[
V =
\begin{bmatrix}
-0.8873 & 0.6933 & 0.0898 \\
0.2839 & 0.4437 & -0.6482 \\
0.3634 & 0.5679 & -0.7561
\end{bmatrix}
\]

Now the eigenvectors are well behaved and \( \text{cond}(V) = 1.4421 \). The ill conditioning is concentrated in the scaling matrix; \( \text{cond}(T) = 8192 \).

This example is small and not really badly scaled, so the computed eigenvalues of \( A \) and \( B \) agree within roundoff error; balancing has little effect on the computed results.

**Algorithm**

**Inputs of Type Double**

For inputs of type double, balance uses the linear algebra package (LAPACK) routines `DGEBAL` (real) and `ZGEBAL` (complex). If you request the output \( T \), balance also uses the LAPACK routines `DGEBAK` (real) and `ZGEBAK` (complex).

**Inputs of Type Single**

For inputs of type single, balance uses the LAPACK routines `SGEBAL` (real) and `CGEBAL` (complex). If you request the output \( T \), balance also uses the LAPACK routines `SGEBAK` (real) and `CGEBAK` (complex).

**Limitations**

Balancing can destroy the properties of certain matrices; use it with some care. If a matrix contains small elements that are due to roundoff error, balancing may scale them up to make them as significant as the other elements of the original matrix.

**See Also**

eig

**References**


2-182
Bar graph (vertical and horizontal)

**Syntax**

- `bar(Y)`
- `bar(x,Y)`
- `bar(...,width)`
- `bar(...,'style')` or `bar(...,'bar_color')`
- `bar(axes_handle,...)`
- `h = bar(...)`
- `hpatches = bar('v6',...)`

- `barh(...)`
- `h = barh(...)`
- `hpatches = barh('v6',...)`

**Description**

A bar graph displays the values in a vector or matrix as horizontal or vertical bars.

`bar(Y)` draws one bar for each element in `Y`. If `Y` is a matrix, `bar` groups the bars produced by the elements in each row. The x-axis scale ranges from 1 to `length(Y)` when `Y` is a vector, and 1 to `size(Y,1)`, which is the number of rows, when `Y` is a matrix.

`bar(x,Y)` draws a bar for each element in `Y` at locations specified in `x`, where `x` is a monotonically increasing vector defining the x-axis intervals for the vertical bars. If `Y` is a matrix, `bar` groups the elements of each row in `Y` at corresponding locations in `x`.

`bar(...,width)` sets the relative bar width and controls the separation of bars within a group. The default width is 0.8, so if you do not specify `x`, the bars within a group have a slight separation. If `width` is 1, the bars within a group touch one another.

`bar(...,'style')` specifies the style of the bars. `'style'` is `'grouped'` or `'stacked'`. `'group'` is the default mode of display.

- `'grouped'` displays `m` groups of `n` vertical bars, where `m` is the number of rows and `n` is the number of columns in `Y`. The group contains one bar per column in `Y`. 
• ‘stacked’ displays one bar for each row in Y. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar(...,'bar_color')` displays all bars using the color specified by the single-letter abbreviation ‘r’, ‘g’, ‘b’, ‘c’, ‘m’, ‘y’, ‘k’, or ‘w’.

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

`h = bar(...)` returns a vector of handles to barseries graphics objects. `bar` creates one barseries graphics object per column in Y.

`barh(...)` and `h = barh(...)` create horizontal bars. Y determines the bar length. The vector x is a monotonic vector defining the y-axis intervals for horizontal bars.

**Backward Compatible Versions**

`hpatches = bar('v6',...)` and `hpatches = barh('v6',...)` return the handles of patch objects instead of barseries objects for compatibility with MATLAB 6.5 and earlier. See patch object properties for a discussion of the properties you can set to control the appearance of these bar graphs.

See Plot Objects and Backward Compatibility for more information.

**Barseries Objects**

Creating a bar graph of an m-by-n matrix creates m groups of n barseries objects. Each barseries objects contains the data for corresponding x values of each bar group (as indicated by the coloring of the bars).

Note that some barseries objects properties set on an individual barseries object, set the values for all barseries objects in the graph. See the property descriptions for information on specific properties.

**Examples**

**Single Series of Data**

This example plots a bell-shaped curve as a bar graph and sets the colors of the bars to red.

```matlab
x = -2.9:0.2:2.9;
```
Bar Graph Options
This example illustrates some bar graph options.

```matlab
Y = round(rand(5,3)*10);
subplot(2,2,1)
bar(Y,'group')
title 'Group'

subplot(2,2,2)
bar(Y,'stack')
title 'Stack'

subplot(2,2,3)
barh(Y,'stack')
title 'Stack'

subplot(2,2,4)
bar(Y,1.5)
title 'Width = 1.5'
```

bar, barh
Setting Properties with Multiobject Graphs

This example creates a graph that displays three groups of bars and contains five barseries objects. Since all barseries objects in a graph share the same baseline, you can set values using any barseries object’s Baseline property. This example uses the first handle returned in h.

\[
Y = \text{randn}(3,5);
\]
\[
h = \text{bar}(Y);
\]
\[
\text{set(get(h(1),'Baseline')),'LineWidth',2,'LineStyle',':')
\]
\[
\text{colormap summer} \ % \text{Change the color scheme}
\]
See Also

bar3, ColorSpec, patch, stairs, hist

“Area, Bar, and Pie Plots” for related functions

“Barseries Properties” on page 2-192

Bar and Area Graphs for more examples
bar3, bar3h

Purpose
Three-dimensional bar chart

Syntax
bar3(Y)
bar3(x,Y)
bar3(...,width)
bar3(...,'style')
bar3(...,LineSpec)
bar3(axes_handle,...)
h = bar3(...)

bar3h(...)
h = bar3h(...)

Description
bar3 and bar3h draw three-dimensional vertical and horizontal bar charts.

bar3(Y) draws a three-dimensional bar chart, where each element in Y corresponds to one bar. 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 size(Y,2), which is the number of columns, and the elements in each row are grouped together.

bar3(x,Y) draws a bar chart of the elements in Y at the locations specified in x, where x is a monotonic vector defining the y-axis intervals for vertical bars. If Y is a matrix, bar3 clusters elements from the same row in Y at locations corresponding to an element in x. Values of elements in each row are grouped together.

bar3(...,width) sets the width of the bars and controls the separation of bars within a group. The default width is 0.8, so if you do not specify x, bars within a group have a slight separation. If width is 1, the bars within a group touch one another.

bar3(...,'style') specifies the style of the bars. 'style' is 'detached', 'grouped', or 'stacked'. 'detached' is the default mode of display.

- 'detached' displays the elements of each row in Y as separate blocks behind one another in the x direction.
• 'grouped' displays $n$ groups of $m$ vertical bars, where $n$ is the number of rows and $m$ is the number of columns in $Y$. The group contains one bar per column in $Y$.

• 'stacked' displays one bar for each row in $Y$. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

$\text{bar3}(\ldots, \text{LineSpec})$ displays all bars using the color specified by LineSpec.

$\text{bar3}(\text{axes\_handles}, \ldots)$ plots into the axes with handle axes\_handle instead of the current axes (gca).

$h = \text{bar3}(\ldots)$ returns a vector of handles to patch graphics objects. $\text{bar3}$ creates one patch object per column in $Y$.

$\text{bar3h}(\ldots)$ and $h = \text{bar3h}(\ldots)$ create horizontal bars. $Y$ determines the bar length. The vector $x$ is a monotonic vector defining the $y$-axis intervals for horizontal bars.

**Examples**

This example creates six subplots showing the effects of different arguments for $\text{bar3}$. The data $Y$ is a seven-by-three matrix generated using the cool colormap:

```matlab
Y = cool(7);
subplot(3,2,1)
bar3(Y,'detached')
title('Detached')

subplot(3,2,2)
bar3(Y,0.25,'detached')
title('Width = 0.25')

subplot(3,2,3)
bar3(Y,'grouped')
title('Grouped')

subplot(3,2,4)
bar3(Y,0.5,'grouped')
title('Width = 0.5')
```
bar3, bar3h

```matlab
subplot(3,2,5)
bar3(Y,'stacked')
title('Stacked')

subplot(3,2,6)
bar3(Y,0.3,'stacked')
title('Width = 0.3')

colormap([1 0 0;0 1 0;0 0 1])
```

**Purpose**  Three-dimensional bar chart
See Also
bar, LineSpec, patch

“Area, Bar, and Pie Plots” for related functions
Bar and Area Graphs for more examples
Barseries Properties

Modifying Properties
You can set and query graphics object properties using the set and get commands or the Property Editor (propertyeditor).

Note that you cannot define default properties for barseries objects.

See Plot Objects for more information on barseries objects.

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

BarLayout
{grouped} | stacked
Specify grouped or stacked bars. Grouped bars display \( m \) groups of \( n \) vertical bars, where \( m \) is the number of rows and \( n \) is the number of columns in the input argument \( Y \). The group contains one bar per column in \( Y \).

Stacked bars display one bar for each row in the input argument \( Y \). The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

BarWidth
scalar in range [0 1]
Width of individual bars. BarWidth specifies the relative bar width and controls the separation of bars within a group. The default width is 0.8, so if you do not specify \( x \), the bars within a group have a slight separation. If width is 1, the bars within a group touch one another.

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 bar graph, obtain the handle of the baseline from the barseries object, and then set line properties that make the baseline a dashed, red line.

```matlab
bar_handle = bar(randn(10,1));
baseline_handle = get(bar_handle,'BaseLine');
set(baseline_handle,'LineStyle','--','Color','red')
```

BaseValue
double: \( y \)-axis value
Value where baseline is drawn. You can specify the value along the \( y \)-axis (vertical bars) or \( x \)-axis (horizontal bars) at which MATLAB draws the baseline.
**Barseries Properties**

**BeingDeleted**  
*on | {off} Read Only*  
This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB 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 barseries 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.
Barseries Properties

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

**Children**

array of graphics object handles

*Children of the barseries object.* The handle of a patch object that is the child of the barseries 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 `Children` property unless you set the root `ShowHiddenHandles` property to `on`:

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

**Clipping**

{on} | off

*Clipping mode.* MATLAB clips bar graphs to the axes plot box by default. If you set `Clipping` to `off`, bars may 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 barseries object. You must specify the callback during the creation of the object. For example,

```
bar(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other barseries properties. Setting this property on an existing barseries 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 barseries object is deleted (e.g., this might happen when you issue a `delete` command on the barseries 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 uses this text for labels for any barseries objects appearing in these legends.

**EdgeColor**

*{[0 0 0]} | none | ColorSpec*

*Color of the edge of the bars.* You can set the color of the edge of the bars to a three-element RGB vector or one of the MATLAB predefined names, including the string `none`. The default edge color is black. See `ColorSpec` for more information on specifying color.

**EraseMode**

*{normal} | none | xor | background*

*Erase mode.* This property controls the technique MATLAB uses to draw and erase bar child objects (the patch object used to construct the bar 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.

**FaceColor**  
{flat} | none | ColorSpec

*Color of filled areas.* 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 all filled areas. See ColorSpec for more information on specifying color.
- **none** — Do not draw faces. Note that EdgeColor is drawn independently of FaceColor.
- **flat** — The color of the filled areas is determined by the figure colormap. See colormap for information on setting the colormap.

**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 barseries object.
Barseries Properties

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

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

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

Overriding Handle Visibility
You can set the Root ShowHiddenHandles property to on to make all handles visible regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties). See also findall.

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 barseries object can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click on the objects that
compose the bar graph. If HitTest is off, clicking the barseries object selects the object below it (which is usually the axes containing it).

**HitTestArea**

`on` | `{off}

*Select barseries object on bars or area of extent.* This property enables you to select barseries objects in two ways:

- Select by clicking bars (default).
- Select by clicking anywhere in the extent of the bar graph.

When HitTestArea is off, you must click the bars to select the barseries object. When HitTestArea is on, you can select the barseries object by clicking anywhere within the extent of the bar graph (i.e., anywhere within a rectangle that encloses all the bars).

**Interruptible**

`{on}` | `off`

*Callback routine interruption mode.* The Interruptible property controls whether a barseries 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 bar property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**LineStyle**

`{−}` | `−−` | `:` | `−.` | `none`

*Line style.* This property specifies the line style used for the bar edges. 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>
</tbody>
</table>
Barseries Properties

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>- .</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

**LineWidth**  
*The width of the bar edges.* Specify this value in points (1 point = 1/72 inch). The default LineWidth is 0.5 points.

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

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

**ShowBaseline**  
*Turn baseline display on or off.* This property determines whether bar plots display a baseline from which the bars are drawn. By default, the baseline is displayed.

**Tag**  
*User-specified object label.* The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when 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 barseries object and set the `Tag` property:

```matlab
t = bar(Y, 'Tag', 'bar1')
```

When you want to access the barseries object, you can use `findobj` to find the barseries object’s handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `bar1`.

```matlab
set(findobj('Tag','bar1'),'FaceColor','red')
```

**Type**

string (read only)

*Type of graphics object.* This property contains a string that identifies the class of the graphics object. For barseries 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 barseries object.* Assign this property the handle of a uicontextmenu object created in the barseries object’s parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the area object.

**UserData**

array

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

**Visible**

{on} | off

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

**XData**

array

*Location of bars.* The x-axis intervals for the vertical bars or y-axis intervals for horizontal bars (as specified by the x input argument). If `YData` is a vector,
XData must be the same size. If YData is a matrix, the length of XData must be equal to the number of rows in YData.

**XDataMode**  
(auto) | manual

*Use automatic or user-specified x-axis values.* If you specify XData (by setting the XData property or specifying the x input argument), MATLAB sets this property to manual.

If you set XDataMode to auto after having specified XData, MATLAB resets the bar locations and x-tick labels (y-tick labels for horizontal bars) to the indices of the YData.

**XDataSource**  
string (MATLAB variable)

*Link XData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

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

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

See the refreshdata reference page for more information.

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

**YData**  
scalar, vector, or matrix

*Bar plot data.* YData contains the data plotted as bars (the Y input argument). Each value in YData is represented by a bar in the bar graph. If YData is a matrix, the bar function creates a “group” or a “stack” of bars for each column in the matrix. See “Bar Graph Options” for examples of grouped and stacked bar graphs.

The input argument Y in the bar function calling syntax assigns values to YData.
**YDataSource**  
string (MATLAB variable)  

Link YData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

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

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

See the refreshdata reference page for more information.

---

**Note** If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Base to decimal number conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>d = base2dec('strn', base)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>d = base2dec('strn', base)</code> converts the string number <code>strn</code> of the specified base into its decimal (base 10) equivalent. <code>base</code> must be an integer between 2 and 36. If <code>'strn'</code> is a character array, each row is interpreted as a string in the specified base.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>The expression <code>base2dec('212', 3)</code> converts <code>212_3</code> to decimal, returning 23.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td><code>dec2base</code></td>
</tr>
</tbody>
</table>
beep

**Purpose**
Produce a beep sound

**Syntax**
beep
beep on
beep off
s = beep

**Description**
beep produces your computer's default beep sound.

beep on turns the beep on.

beep off turns the beep off.

s = beep returns the current beep mode (on or off).
**Purpose**

Bessel function of the third kind (Hankel function)

**Syntax**

- \( H = \text{besselh}(\nu,K,Z) \)
- \( H = \text{besselh}(\nu,Z) \)
- \( H = \text{besselh}(\nu,K,Z,1) \)
- \([H,\text{ierr}] = \text{besselh}(\ldots)\)

**Definitions**

The differential equation

\[
\frac{d^2}{dz^2} y + \frac{d}{dz} \left( z^2 - \nu^2 \right) y = 0
\]

where \( \nu \) is a nonnegative constant, is called **Bessel’s equation**, and its solutions are known as **Bessel functions**. \( J_\nu(z) \) and \( J_{-\nu}(z) \) form a fundamental set of solutions of Bessel’s equation for noninteger \( \nu \). \( Y_\nu(z) \) is a second solution of Bessel’s equation – linearly independent of \( J_\nu(z) \) – defined by

\[
Y_\nu(z) = \frac{J_\nu(z) \cos(\nu \pi) - J_{-\nu}(z)}{\sin(\nu \pi)}
\]

The relationship between the Hankel and Bessel functions is

\[
H^{(1)}_\nu(z) = J_\nu(z) + i Y_\nu(z)
\]

\[
H^{(2)}_\nu(z) = J_\nu(z) - i Y_\nu(z)
\]

where \( J_\nu(z) \) is \text{besselj}, and \( Y_\nu(z) \) is \text{bessely}.

**Description**

- \( H = \text{besselh}(\nu,K,Z) \) computes the Hankel function \( H^{(K)}_\nu(z) \), where \( K = 1 \) or 2, for each element of the complex array \( Z \). If \( \nu \) and \( Z \) are arrays of the same size, the result is also that size. If either input is a scalar, \text{besselh} expands it to the other input’s size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

- \( H = \text{besselh}(\nu,Z) \) uses \( K = 1 \).

- \( H = \text{besselh}(\nu,K,Z,1) \) scales \( H^{(K)}_\nu(z) \) by \( \exp(-i*Z) \) if \( K = 1 \), and by \( \exp(+i*Z) \) if \( K = 2 \).
[H,ierr] = besselh(...) also returns completion flags in an array the same size as H.

<table>
<thead>
<tr>
<th>ierr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>besselh successfully computed the Hankel function for this element.</td>
</tr>
<tr>
<td>1</td>
<td>Illegal arguments.</td>
</tr>
<tr>
<td>2</td>
<td>Overflow. Returns Inf.</td>
</tr>
<tr>
<td>3</td>
<td>Some loss of accuracy in argument reduction.</td>
</tr>
<tr>
<td>4</td>
<td>Unacceptable loss of accuracy, Z or nu too large.</td>
</tr>
<tr>
<td>5</td>
<td>No convergence. Returns NaN.</td>
</tr>
</tbody>
</table>

**Examples**

This example generates the contour plots of the modulus and phase of the Hankel function $H_0^{(1)}(z)$ shown on page 359 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

It first generates the modulus contour plot

```matlab
[X,Y] = meshgrid(-4:0.025:2,-1.5:0.025:1.5);
H = besselh(0,1,X+i*Y);
contour(X,Y,abs(H),0:0.2:3.2), hold on
```
then adds the contour plot of the phase of the same function.

```matlab
contour(X,Y,(180/pi)*angle(H),-180:10:180); hold off
```

**See Also**
besselj, bessely, besseli, besselk
References

Purpose
Modified Bessel function of the first kind

Syntax
I = besseli(nu,Z)
I = besseli(nu,Z,1)
[I,ierr] = besseli(...)

Definitions
The differential equation
\[ z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} - (z^2 + \nu^2)y = 0 \]
where \( \nu \) is a real constant, is called the modified Bessel's equation, and its
solutions are known as modified Bessel functions.

\( I_\nu(z) \) and \( K_\nu(z) \) form a fundamental set of solutions of the modified Bessel's
equation for noninteger \( \nu \). \( I_\nu(z) \) is defined by

\[ I_\nu(z) = \left( \frac{z}{2} \right)^\nu \sum_{k=0}^{\infty} \frac{\left( \frac{z^2}{4} \right)^k}{k! \Gamma(\nu+k+1)} \]

where \( \Gamma(a) \) is the gamma function.

\( K_\nu(z) \) is a second solution, independent of \( I_\nu(z) \). It can be computed using
besselk.

Description
\( I = \text{besseli}(\nu,Z) \) computes the modified Bessel function of the first kind,
\( I_\nu(z) \), for each element of the array \( Z \). The order \( \nu \) need not be an integer, but
must be real. The argument \( Z \) can be complex. The result is real where \( Z \) is
positive.

If \( \nu \) and \( Z \) are arrays of the same size, the result is also that size. If either input
is a scalar, it is expanded to the other input's size. If one input is a row vector
and the other is a column vector, the result is a two-dimensional table of
function values.

\( I = \text{besseli}(\nu,Z,1) \) computes \( \text{besseli}(\nu,Z) \cdot \exp(-\abs{\text{real}(Z)}) \).  

2-209
besseli

[I, ierr] = besseli(...) also returns completion flags in an array the same size as I.

<table>
<thead>
<tr>
<th>ierr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>besseli successfully computed the modified Bessel function for this element.</td>
</tr>
<tr>
<td>1</td>
<td>Illegal arguments.</td>
</tr>
<tr>
<td>2</td>
<td>Overflow. Returns Inf.</td>
</tr>
<tr>
<td>3</td>
<td>Some loss of accuracy in argument reduction.</td>
</tr>
<tr>
<td>4</td>
<td>Unacceptable loss of accuracy, Z or nu too large.</td>
</tr>
<tr>
<td>5</td>
<td>No convergence. Returns NaN.</td>
</tr>
</tbody>
</table>

**Examples**

**Example 1.**

```matlab
format long
z = (0:0.2:1)';
besseli(1,z)
```

```
ans =
   0
0.10050083402813
0.20402675573357
0.31370402560492
0.43286480262064
0.56515910399249
```

**Example 2.** `besseli(3:9,(0:.2,10)',1)` generates the entire table on page 423 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

**Algorithm**

The besseli functions uses a Fortran MEX-file to call a library developed by D. E. Amos [3] [1].

**See Also**

airy, besselh, besselj, besselk, bessely
References


**Purpose**

Bessel function of the first kind

**Syntax**

\[ \text{J} = \text{besselj}(\nu, Z) \]

\[ \text{J} = \text{besselj}(\nu, Z, 1) \]

\[ [\text{J}, \text{ierr}] = \text{besselj}(\nu, Z) \]

**Definition**

The differential equation

\[
\frac{z^2}{\nu^2} \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + (z^2 - \nu^2)y = 0
\]

where \( \nu \) is a real constant, is called **Bessel’s equation**, and its solutions are known as **Bessel functions**.

\( J_\nu(z) \) and \( Y_\nu(z) \) form a fundamental set of solutions of Bessel’s equation for noninteger \( \nu \). \( J_\nu(z) \) is defined by

\[
J_\nu(z) = \left( \frac{z}{\nu} \right)^\nu \sum_{k=0}^{\infty} \frac{\left( -\frac{z^2}{4} \right)^k}{k! \Gamma(\nu + k + 1)}
\]

where \( \Gamma(a) \) is the gamma function.

\( Y_\nu(z) \) is a second solution of Bessel’s equation that is linearly independent of \( J_\nu(z) \). It can be computed using \text{bessely}.

**Description**

\( J = \text{besselj}(\nu, Z) \) computes the Bessel function of the first kind, \( J_\nu(z) \), for each element of the array \( Z \). The order \( \nu \) need not be an integer, but must be real. The argument \( Z \) can be complex. The result is real where \( Z \) is positive.

If \( \nu \) and \( Z \) are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input’s size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

\( J = \text{besselj}(\nu, Z, 1) \) computes \( \text{besselj}(\nu, Z) \cdot \exp(-\text{abs}(\text{imag}(Z))) \).

\[ [J, \text{ierr}] = \text{besselj}(\nu, Z) \) also returns completion flags in an array the same size as \( J \).
The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind, \( besselh \), \( besselj \), and \( bessely \). The Hankel functions also form a fundamental set of solutions to Bessel's equation (see \( besselh \)).

### Examples

**Example 1.**

```matlab
global format long
z = (0:0.2:1)';
besselj(1,z)
```

```
ans =
0
0.09950083263924
0.19602657795532
0.28670098806392
0.36884204609417
0.44005058574493
```

### Remarks

<table>
<thead>
<tr>
<th>ierr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( besselj ) successfully computed the Bessel function for this element.</td>
</tr>
<tr>
<td>1</td>
<td>Illegal arguments.</td>
</tr>
<tr>
<td>2</td>
<td>Overflow. Returns Inf.</td>
</tr>
<tr>
<td>3</td>
<td>Some loss of accuracy in argument reduction.</td>
</tr>
<tr>
<td>4</td>
<td>Unacceptable loss of accuracy, ( Z ) or ( n )u too large.</td>
</tr>
<tr>
<td>5</td>
<td>No convergence. Returns NaN.</td>
</tr>
</tbody>
</table>

\[
H^{(1)}_\nu(z) = J_\nu(z) + i Y_\nu(z)
\]

\[
H^{(2)}_\nu(z) = J_\nu(z) - i Y_\nu(z)
\]

where \( H^{(K)}_\nu(z) \) is \( besselh \), \( J_\nu(z) \) is \( besselj \), and \( Y_\nu(z) \) is \( bessely \). The Hankel functions also form a fundamental set of solutions to Bessel's equation (see \( besselh \)).
Example 2. besselj(3:9,(0:.2:10)') generates the entire table on page 398 of [1] Abramowitz and Stegun, Handbook of Mathematical Functions.

Algorithm

The besselj function uses a Fortran MEX-file to call a library developed by D. E. Amos [3] [4].

See Also

besselh, besseli, besselk, bessely

References


**Purpose**
Modified Bessel function of the second kind

**Syntax**

\[
K = \text{besselk}(\nu,Z) \\
K = \text{besselk}(\nu,Z,1) \\
[K,ierr] = \text{besselk}(...) 
\]

**Definitions**
The differential equation

\[
z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} - (z^2 + \nu^2)y = 0
\]

where \( \nu \) is a real constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

A solution \( K_\nu(z) \) of the second kind can be expressed as

\[
K_\nu(z) = \left( \frac{\pi}{2} \right) \frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}
\]

where \( I_{\nu}(z) \) and \( I_{-\nu}(z) \) form a fundamental set of solutions of the modified Bessel's equation for noninteger \( \nu \)

\[
I_{\nu}(z) = \left( \frac{z}{2} \right)^{\nu} \sum_{k=0}^{\infty} \frac{\left( \frac{z^2}{4} \right)^k}{k! \Gamma(\nu + k + 1)}
\]

and \( \Gamma(a) \) is the gamma function. \( K_\nu(z) \) is independent of \( I_\nu(z) \).

\( I_\nu(z) \) can be computed using \( \text{besseli} \).

**Description**

\( K = \text{besselk}(\nu,Z) \) computes the modified Bessel function of the second kind, \( K_\nu(z) \), for each element of the array \( Z \). The order \( \nu \) need not be an integer, but must be real. The argument \( Z \) can be complex. The result is real where \( Z \) is positive.

If \( \nu \) and \( Z \) are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.
besselk

K = besselk(nu,Z,1) computes \( besselk(nu,Z) \cdot \exp(Z) \).

\([K,ierr] = \text{besselk}(\ldots)\) also returns completion flags in an array the same size as \( K \).

<table>
<thead>
<tr>
<th>ierr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\text{besselk} succesfuly computed the modified Bessel function for this element.</td>
</tr>
<tr>
<td>1</td>
<td>Illegal arguments.</td>
</tr>
<tr>
<td>2</td>
<td>Overflow. Returns ( \text{Inf} ).</td>
</tr>
<tr>
<td>3</td>
<td>Some loss of accuracy in argument reduction.</td>
</tr>
<tr>
<td>4</td>
<td>Unacceptable loss of accuracy, ( Z ) or ( nu ) too large.</td>
</tr>
<tr>
<td>5</td>
<td>No convergence. Returns ( \text{NaN} ).</td>
</tr>
</tbody>
</table>

\textbf{Examples}

\textbf{Example 1.}

```
format long
z = (0:0.2:1)';

besselk(1,z)
```

\text{ans =}

\[
\begin{array}{c}
\text{Inf} \\
4.77597254322047 \\
2.18435442473269 \\
1.30283493976350 \\
0.86178163447218 \\
0.60190723019723
\end{array}
\]

\textbf{Example 2.} \text{besselk}(3:9,(0:.2:10)',1) generates part of the table on page 424 of \cite{Abramowitz} Abramowitz and Stegun, \textit{Handbook of Mathematical Functions}.

\textbf{Algorithm}

The \text{besselk} function uses a Fortran MEX-file to call a library developed by D. E. Amos \cite{Amos3} \cite{Amos4}.
See Also
airy, besselh, besseli, besselj, bessely

References


**Purpose**

Bessel functions of the second kind

**Syntax**

\[ Y = \text{bessely}(\nu, Z) \]

\[ Y = \text{bessely}(\nu, Z, 1) \]

\[ [Y, ierr] = \text{bessely}(\nu, Z) \]

**Definition**

The differential equation

\[ z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + (z^2 - \nu^2)y = 0 \]

where \( \nu \) is a real constant, is called Bessel’s equation, and its solutions are known as Bessel functions.

A solution \( Y_\nu(z) \) of the second kind can be expressed as

\[ Y_\nu(z) = \frac{J_\nu(z) \cos(\nu \pi) - J_{-\nu}(z)}{\sin(\nu \pi)} \]

where \( J_\nu(z) \) and \( J_{-\nu}(z) \) form a fundamental set of solutions of Bessel’s equation for noninteger \( \nu \)

\[ J_\nu(z) = \left( \frac{z}{2} \right)^\nu \sum_{k=0}^{\infty} \frac{\left( -\frac{z^2}{4} \right)^k}{k! \Gamma(k + \nu + 1)} \]

and \( \Gamma(a) \) is the gamma function. \( Y_\nu(z) \) is linearly independent of \( J_\nu(z) \)

\( J_\nu(z) \) can be computed using \text{besselj}.

**Description**

\( Y = \text{bessely}(\nu, Z) \) computes Bessel functions of the second kind, \( Y_\nu(z) \), for each element of the array \( Z \). The order \( \nu \) need not be an integer, but must be real. The argument \( Z \) can be complex. The result is real where \( Z \) is positive.

If \( \nu \) and \( Z \) are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input’s size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

\( Y = \text{bessely}(\nu, Z, 1) \) computes \( \text{bessely}(\nu, Z) . \exp(-\text{abs} \left( \text{imag}(Z) \right)) \).
[Y,ierr] = bessely(nu,Z) also returns completion flags in an array the same size as Y.

<table>
<thead>
<tr>
<th>ierr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bessely successfully computed the Bessel function for this element.</td>
</tr>
<tr>
<td>1</td>
<td>Illegal arguments.</td>
</tr>
<tr>
<td>2</td>
<td>Overflow. Returns Inf.</td>
</tr>
<tr>
<td>3</td>
<td>Some loss of accuracy in argument reduction.</td>
</tr>
<tr>
<td>4</td>
<td>Unacceptable loss of accuracy, Z or nu too large.</td>
</tr>
<tr>
<td>5</td>
<td>No convergence. Returns NaN.</td>
</tr>
</tbody>
</table>

Remarks

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind,

\[ H^{(1)}_\nu(z) = J_\nu(z) + i Y_\nu(z) \]
\[ H^{(2)}_\nu(z) = J_\nu(z) - i Y_\nu(z) \]

where \( H^{(K)}_\nu(z) \) is besselh, \( J_\nu(z) \) is besselj, and \( Y_\nu(z) \) is bessely. The Hankel functions also form a fundamental set of solutions to Bessel’s equation (see besselh).

Examples

Example 1.

```matlab
format long
z = (0:0.2:1) ’;

bessely(1,z)
```

```
ans =
   -Inf
-3.32382498811185
-1.78087204427005
```

bessely

-1.26039134717739
-0.97814417668336
-0.78121282130029

Example 2. bessely(3:9, (0:.2:10)' ) generates the entire table on page 399 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithm

The bessely function uses a Fortran MEX-file to call a library developed by D. E Amos [3] [4].

See Also

besselh, besseli, besselj, besselk

References


Purpose

Beta function

Syntax

\[ B = \text{beta}(Z,W) \]

Definition

The beta function is

\[ B(z, w) = \int_0^1 t^{z-1}(1-t)^{w-1} \, dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \]

where \( \Gamma(z) \) is the gamma function.

Description

\( B = \text{beta}(Z,W) \) computes the beta function for corresponding elements of arrays \( Z \) and \( W \). The arrays must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

In this example, which uses integer arguments,

\[
\begin{align*}
\text{beta}(n,3) &= (n-1)! \times 2! / (n+2)! \\
&= 2 / (n \times (n+1) \times (n+2))
\end{align*}
\]

is the ratio of fairly small integers, and the rational format is able to recover the exact result.

\[
\text{format rat} \\
\text{beta}((0:10)',3)
\]

\[
\text{ans} =
\begin{align*}
1/0 \\
1/3 \\
1/12 \\
1/30 \\
1/60 \\
1/105 \\
1/168 \\
1/252 \\
1/360 \\
1/495 \\
1/660
\end{align*}
\]
**beta**

**Algorithm**

\[ \text{beta}(z, w) = \exp(\text{gammaln}(z)+\text{gammaln}(w)-\text{gammaln}(z+w)) \]

**See Also**

betainc, betaln, gammaln
Purpose
Incomplete beta function

Syntax
I = betainc(X,Z,W)

Definition
The incomplete beta function is
\[ I_x(z, w) = \frac{1}{B(z, w)} \int_0^x t^{z-1}(1-t)^{w-1} \, dt \]
where \( B(z, w) \), the beta function, is defined as
\[ B(z, w) = \int_0^1 t^{z-1}(1-t)^{w-1} \, dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \]
and \( \Gamma(z) \) is the gamma function.

Description
\( I = \text{betainc}(X,Z,W) \) computes the incomplete beta function for corresponding elements of the arrays \( X, Z \) and \( W \). The elements of \( X \) must be in the closed interval \([0,1]\). The arrays \( Z \) and \( W \) must be nonnegative and real. All arrays must be the same size, or any of them can be scalar.

Examples
format long
betainc(.5,(0:10)',3)

ans =
1.00000000000000
0.87500000000000
0.68750000000000
0.50000000000000
0.34375000000000
0.22656250000000
0.14453125000000
0.08984375000000
0.05468750000000
0.03271484375000
0.01928710937500

See Also
beta, betaln
**betaln**

**Purpose**
Logarithm of beta function

**Syntax**
L = betaln(Z,W)

**Description**
L = betaln(Z,W) computes the natural logarithm of the beta function \( \log(\text{beta}(Z,W)) \), for corresponding elements of arrays Z and W, without computing \( \text{beta}(Z,W) \). Since the beta function can range over very large or very small values, its logarithm is sometimes more useful.

Z and W must be real and nonnegative. They must be the same size, or either can be scalar.

**Examples**
```matlab
x = 510
betaln(x,x)
```
```
ans =
-708.8616
```
-708.8616 is slightly less than \( \log(\text{realmin}) \). Computing \( \text{beta}(x,x) \) directly would underflow (or be denormal).

**Algorithm**
\[ \text{betaln}(z,w) = \text{gammaln}(z)+\text{gammaln}(w)-\text{gammaln}(z+w) \]

**See Also**
beta, betainc, gammaln
Purpose

BiConjugate Gradients method

Syntax

\[ x = \text{bicg}(A,b) \]
\[ \text{bicg}(A,b,tol) \]
\[ \text{bicg}(A,b,tol,maxit) \]
\[ \text{bicg}(A,b,tol,maxit,M) \]
\[ \text{bicg}(A,b,tol,maxit,M1,M2) \]
\[ \text{bicg}(A,b,tol,maxit,M1,M2,x0) \]
\[ \text{bicg}(afun,b,tol,maxit,mfun1,mfun2,x0,p1,p2,...) \]
\[ [x,\text{flag}] = \text{bicg}(A,b,...) \]
\[ [x,\text{flag},\text{relres}] = \text{bicg}(A,b,...) \]
\[ [x,\text{flag},\text{relres},\text{iter}] = \text{bicg}(A,b,...) \]
\[ [x,\text{flag},\text{relres},\text{iter},\text{resvec}] = \text{bicg}(A,b,...) \]

Description

\[ x = \text{bicg}(A,b) \] attempts to solve the system of linear equations \( A \cdot x = b \) for \( x \). The \( n \)-by-\( n \) coefficient matrix \( A \) must be square and should be large and sparse. The column vector \( b \) must have length \( n \). \( A \) can be a function \( afun \) such that \( afun(x) \) returns \( A \cdot x \) and \( afun(x,'\text{transp}') \) returns \( A^\prime \cdot x \).

If \text{bicg} converges, it displays a message to that effect. If \text{bicg} fails to converge after the maximum number of iterations or halts for any reason, it prints a warning message that includes the relative residual \( \frac{\text{norm}(b-A \cdot x)}{\text{norm}(b)} \) and the iteration number at which the method stopped or failed.

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

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

\text{bicg}(A,b,tol,maxit,M) and \text{bicg}(A,b,tol,maxit,M1,M2) use the preconditioner \( M \) 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{bicg} applies no preconditioner. \( M \) can be a function \( mfun \) such that \( mfun(x) \) returns \( M \cdot x \) and \( mfun(x,'\text{transp}') \) returns \( M^\prime \cdot x \).

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

bicg(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] = bicg(A,b,...) also returns a convergence flag.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bicg converged to the desired tolerance tol within maxit iterations.</td>
</tr>
<tr>
<td>1</td>
<td>bicg 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>bicg stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during bicg 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] = bicg(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

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

[x,flag,relres,iter,resvec] = bicg(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;
```
bicg

maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = bicg(A,b,tol,maxit,M1,M2,[]);

displays this message
bicg converged at iteration 9 to a solution with relative residual 5.3e-009

Alternatively, use this matrix-vector product function

```matlab
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 bicg.

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

**Example 2.** This examples demonstrates the use of a preconditioner. Start with \( A = \text{west0479} \), a real 479-by-479 sparse matrix, and define \( b \) so that the true solution is a vector of all ones.

```matlab
load west0479;
A = west0479;
b = sum(A,2);
```

You can accurately solve \( A^x = b \) using backslash since \( A \) is not so large.

```matlab
x = A \ b;
norm(b-A*x) / norm(b)
```

ans =

8.3154e-017
Now try to solve \( A \times x = b \) with \texttt{bicg}.

\[
[x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \texttt{bicg}(A,b)
\]

<table>
<thead>
<tr>
<th>flag</th>
<th>relres</th>
<th>iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The value of \texttt{flag} indicates that \texttt{bicg} iterated the default 20 times without converging. The value of \texttt{iter} shows that the method behaved so badly that the initial all-zero guess was better than all the subsequent iterates. The value of \texttt{relres} supports this: \( \text{relres} = \frac{\text{norm}(b-A \times x)}{\text{norm}(b)} = \frac{\text{norm}(b)}{\text{norm}(b)} = 1 \).

You can confirm that the unpreconditioned method oscillates rather wildly by plotting the relative residuals at each iteration.

\[
\text{semilogy}(0:20,\text{resvec}/\text{norm}(b),'-o')
\]
\[
\text{xlabel('Iteration Number')}
\]
\[
\text{ylabel('Relative Residual')}
\]
Now, try an incomplete LU factorization with a drop tolerance of $1\times10^{-5}$ for the preconditioner.

\[
[L1, U1] = \text{luinc}(A, 1e^{-5});
\]

Warning: Incomplete upper triangular factor has 1 zero diagonal. It cannot be used as a preconditioner for an iterative method.

\[
nnz(A), \, nnz(L1), \, nnz(U1)
\]

\[
\begin{align*}
\text{ans} &= 1887 \\
\text{ans} &= 5562 \\
\text{ans} &= 4320
\end{align*}
\]

The zero on the main diagonal of the upper triangular $U1$ indicates that $U1$ is singular. If you try to use it as a preconditioner,

\[
[x, \text{flag}, \text{relres}, \text{iter}, \text{resvec}] = \text{bicg}(A, b, 1e^{-6}, 20, L1, U1)
\]

\[
\begin{align*}
\text{flag} &= 2 \\
\text{relres} &= 1 \\
\text{iter} &= 0 \\
\text{resvec} &= 7.0557e+005
\end{align*}
\]

the method fails in the very first iteration when it tries to solve a system of equations involving the singular $U1$ using backslash. $\text{bicg}$ is forced to return the initial estimate since no other iterates were produced.

Try again with a slightly less sparse preconditioner.

\[
[L2, U2] = \text{luinc}(A, 1e^{-6});
\]
nnz(L2), nnz(U2)

ans =
   6231
ans =
   4559

This time U2 is nonsingular and may be an appropriate preconditioner.

[x,flag,relres,iter,resvec] = bicg(A,b,1e-15,10,L2,U2)

flag =
   0
relres =
   2.8664e-016
iter =
   8

and bicg converges to within the desired tolerance at iteration number 8.
Decreasing the value of the drop tolerance increases the fill-in of the incomplete factors but also increases the accuracy of the approximation to the original matrix. Thus, the preconditioned system becomes closer to
\[ \text{inv}(U)\text{inv}(L)^*L^*U^*x = \text{inv}(U)\text{inv}(L)^*b, \]
where L and U are the true LU factors, and closer to being solved within a single iteration.

The next graph shows the progress of bicg using six different incomplete LU factors as preconditioners. Each line in the graph is labeled with the drop tolerance of the preconditioner used in bicg.
See Also  
bicgstab, cgs, gmres, lsqr, luinc, minres, pcg, qmr, symmlq
@ (function handle), \ (backslash)

References  
Purpose

BiConjugate Gradients Stabilized method

Syntax

\[ x = \text{bicgstab}(A,b) \]
\[ \text{bicgstab}(A,b,tol) \]
\[ \text{bicgstab}(A,b,tol,maxit) \]
\[ \text{bicgstab}(A,b,tol,maxit,M) \]
\[ \text{bicgstab}(A,b,tol,maxit,M1,M2) \]
\[ \text{bicgstab}(A,b,tol,maxit,M1,M2,x0) \]
\[ \text{bicgstab}(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...) \]
\[ [x,\text{flag}] = \text{bicgstab}(A,b,...) \]
\[ [x,\text{flag},\text{relres}] = \text{bicgstab}(A,b,...) \]
\[ [x,\text{flag},\text{relres},\text{iter}] = \text{bicgstab}(A,b,...) \]
\[ [x,\text{flag},\text{relres},\text{iter},\text{resvec}] = \text{bicgstab}(A,b,...) \]

Description

\[ x = \text{bicgstab}(A,b) \] attempts to solve the system of linear equations \( A*x = b \) for \( x \). The \( n \)-by-\( n \) coefficient matrix \( A \) must be square and should be large and sparse. The column vector \( b \) must have length \( n \). \( A \) can be a function \( \text{afun} \) such that \( \text{afun}(x) \) returns \( A*x \).

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

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

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

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

\[ \text{bicgstab}(A,b,tol,maxit,M1,M2,x0) \] specifies the initial guess. If \( x0 \) is [], then \text{bicgstab} uses the default, an all zero vector.
bicgstab(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] = bicgstab(A,b,...) also returns a convergence flag.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bicgstab converged to the desired tolerance tol within maxit iterations.</td>
</tr>
<tr>
<td>1</td>
<td>bicgstab 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>bicgstab stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during bicgstab 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] = bicgstab(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = bicgstab(A,b,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit. iter can be an integer + 0.5, indicating convergence half way through an iteration.

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

Example

Example 1. This example first solves Ax = b by providing A and the preconditioner M directly as arguments. It then solves the same system using functions that return A and the preconditioner.

A = gallery('wilk',21);
b = sum(A,2);
bicgstab

```matlab
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = bicgstab(A,b,tol,maxit,M1,[],[]);
```
displays this message

```
bicgstab converged at iteration 12.5 to a solution with relative residual 2.9e-014
```
Alternatively, use this matrix-vector product function

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

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

```matlab
x1 = bicgstab(@afun,b,tol,maxit,@mfun,[],[],21);
```
Note that both afun and mfun must accept bicgstab's extra input n=21.

**Example 2.** This examples demonstrates the use of a preconditioner. Start with \( A = \text{west0479} \), a real 479-by-479 sparse matrix, and define \( b \) so that the true solution is a vector of all ones.

```matlab
load west0479;
A = west0479;
b = sum(A,2);
[x,flag] = bicgstab(A,b)
```
flag is 1 because bicgstab does not converge to the default tolerance 1e-6 within the default 20 iterations.

```matlab
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = bicgstab(A,b,1e-6,20,L1,U1)
```
flag1 is 2 because the upper triangular $U_1$ has a zero on its diagonal. This causes \texttt{bicgstab} to fail in the first iteration when it tries to solve a system such as $U_1 y = r$ using backslash.

$[L_2, U_2] = \text{luinc}(A, 1e-6);$
$[x_2, \text{flag2, relres2, iter2, resvec2}] = \text{bicgstab}(A, b, 1e-15, 10, L_2, U_2)$

flag2 is 0 because \texttt{bicgstab} converges to the tolerance of $3.1757e-016$ (the value of relres2) at the sixth iteration (the value of iter2) when preconditioned by the incomplete LU factorization with a drop tolerance of $1e-6$. resvec2(1) = norm(b) and resvec2(13) = norm(b-A*x2). You can follow the progress of \texttt{bicgstab} by plotting the relative residuals at the halfway point and end of each iteration starting from the initial estimate (iterate number 0).

semilogy(0:0.5:iter2, resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
bicgstab

See Also
bicg, cgs, gmres, lsqr, luinc, minres, pcg, qmr, symmlq
@ (function handle), \ (backslash)

References
Purpose  Binary to decimal number conversion

Syntax  bin2dec(binarystr)

Description  bin2dec(binarystr) interprets the binary string binarystr and returns the equivalent decimal number.

bin2dec ignores any space (‘ ‘) characters in the input string.

Examples  Binary 010111 converts to decimal 23:

```matlab
bin2dec('010111')
an =
    23
```

Because space characters are ignored, this string yields the same result:

```matlab
bin2dec(' 010   111 ')
an =
    23
```

See Also  dec2bin
### binary (ftp)

**Purpose**
Set FTP transfer type to binary.

**Syntax**
```matlab
binary(f)
```

**Description**
`binary(f)` sets the FTP download and upload mode to binary, which does not convert new lines, where `f` was created using `ftp`. Use this function when downloading or uploading any nontext file, such as an executable or ZIP archive.

**Examples**
Connect to the MathWorks FTP server, and display the FTP object.

```matlab
tmw=ftp('ftp.mathworks.com');
disp(tmw)
```

**FTP Object**
- host: `ftp.mathworks.com`
- user: `anonymous`
- dir: `/`
- mode: `binary`

Note that the FTP object defaults to binary mode.

Use the `ascii` function to set the FTP mode to ASCII, and use the `disp` function to display the FTP object.

```matlab
ascii(tmw)
disp(tmw)
```

**FTP Object**
- host: `ftp.mathworks.com`
- user: `anonymous`
- dir: `/`
- mode: `ascii`

Note that the FTP object is now set to ASCII mode.

Use the `binary` function to set the FTP mode to binary, and use the `disp` function to display the FTP object.

```matlab
binary(tmw)
disp(tmw)
```

**FTP Object**
- host: `ftp.mathworks.com`
- user: `anonymous`
Purpose  Bitwise AND

Syntax  \[ C = \text{bitand}(A, B) \]

Description  \( C = \text{bitand}(A, B) \) returns the bitwise AND of two unsigned integer arguments \( A \) and \( B \).

Examples  Example 1
The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bitwise AND on these numbers yields 01001, or 9:

\[
C = \text{bitand}(\text{uint8}(13), \text{uint8}(27))
\]

\[
C = 9
\]

Example 2
Create a truth table for a logical AND operation:

\[
A = \text{uint8}([0 1; 0 1]);
B = \text{uint8}([0 0; 1 1]);
\]

\[
\text{TT} = \text{bitand}(A, B)
\]

\[
\text{TT} =
\begin{array}{cc}
0 & 0 \\
0 & 1 \\
\end{array}
\]

See Also  bitcmp, bitget, bitmax, bitor, bitset, bitshift, bitxor
Purpose

Complement bits

Syntax

C = bitcmp(A, n)

Description

C = bitcmp(A, n) returns the bitwise complement of A as an n-bit unsigned integer.

The value assigned to A may not have any bits set higher than n, (that is, its value may not be greater than $2^n - 1$). If n is the number of bits in the unsigned integer class of A (for example, if A is a uint32 and n is 32) then the value of A may be between 0 and intmax(class(A)).

Example

Example 1

With eight-bit arithmetic, the ones’ complement of 01100011 (99, decimal) is 10011100 (156, decimal).

    C = bitcmp(uint8(99), 8)
    C =
       156

Example 2

find the complement of 255 (hexadecimal FF):

    a = uint16(intmax('uint8'));
    bitcmp(a, 8)
    ans =
       0

See Also

bitand, bitget, bitmax, bitor, bitset, bitshift, bitxor
Purpose
Get bit

Syntax
C = bitget(A, bit)

Description
C = bitget(A, bit) returns the value of the bit at position bit in A. Operand A must be an unsigned integer, and bit must be a number between 1 and the number of bits in the unsigned integer class of A (e.g., 32 for the uint32 class).

Example
Example 1
The dec2bin function converts decimal numbers to binary. However, you can also use the bitget function to show the binary representation of a decimal number. Just test successive bits from most to least significant:

```matlab
disp(dec2bin(13))
1101

C = bitget(uint8(13), 4:-1:1)
C =
    1     1     0     1
```

Example 2
Prove that intmax sets all the bits to 1:

```matlab
a = intmax('uint8');
if all(bitget(a, 1:8))
    disp('All the bits have value 1.')
end

All the bits have value 1.
```

See Also
bitand, bitcmp, bitmax, bitor, bitset, bitshift, bitxor
### Purpose
Maximum floating-point integer

### Syntax
`bitmax`

### Description
`bitmax` returns the maximum unsigned double-precision floating-point integer for your computer. It is the value when all bits are set, namely the value $2^{53}-1$.

**Note** Instead of integer-valued double-precision variables, use unsigned integers for bit manipulations and replace `bitmax` with `intmax`.

### Examples
Display in different formats the largest floating point integer and the largest 32 bit unsigned integer:

```matlab
format long e
bitmax
ans = 9.007199254740991e+015

intmax('uint32')
ans = 4294967295

format hex
bitmax
ans = 433fffffffffffff

intmax('uint32')
ans = ffffffff
```

In the second `bitmax` statement, the last 13 hex digits of `bitmax` are f, corresponding to 52 1's (all 1's) in the mantissa of the binary representation. The first 3 hex digits correspond to the sign bit 0 and the 11 bit biased exponent $10000110011$ in binary ($1075$ in decimal), and the actual exponent is $(1075 - 1023) = 52$. Thus the binary value of `bitmax` is $1.111...111 \times 2^{52}$ with 52 trailing 1's, or $2^{53}-1$. 

2-242
See Also

bitand, bitcmp, bitget, bitor, bitset, bitshift, bitxor
Purpose

Bitwise OR

Syntax

\[ C = \text{bitor}(A, B) \]

Description

\( C = \text{bitor}(A, B) \) returns the bitwise OR of two unsigned integer arguments \( A \) and \( B \).

Examples

Example 1

The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bitwise OR on these numbers yields 11111, or 31.

\[
\begin{align*}
C &= \text{bitor}(\text{uint8}(13), \text{uint8}(27)) \\
C &= 31
\end{align*}
\]

Example 2

Create a truth table for a logical OR operation:

\[
\begin{align*}
A &= \text{uint8}([0 \ 1 ; \ 0 \ 1]) \\
B &= \text{uint8}([0 \ 0 ; \ 1 \ 1])
\end{align*}
\]

\[
\begin{array}{c c}
0 & 1 \\
1 & 1
\end{array}
\]

See Also

bitand, bitcmp, bitget, bitmax, bitset, bitshift, bitxor
Purpose
Set bit

Syntax
C = bitset(A, bit)
C = bitset(A, bit, v)

Description
C = bitset(A, bit) sets bit position bit in A to 1 (on). A must be an unsigned integer and bit must be a number between 1 and the number of bits in the unsigned integer class of A (e.g., 32 for the uint32 class).

C = bitset(A, bit, v) sets the bit at position bit to the value v, which must be either 0 or 1.

Examples
Example 1
Setting the fifth bit in the five-bit binary representation of the integer 9 (01001) yields 11001, or 25:

C = bitset(uint8(9), 5)
C =
25

Example 2
Repeatedly subtract powers of 2 from the largest uint32 value:

a = intmax('uint32')
for k = 1:32
    a = bitset(a, 32-k+1, 0)
end

See Also
bitand, bitcmp, bitget, bitmax, bitor, bitshift, bitxor
bitshift

Purpose
Bitwise shift

Syntax
\[ C = \text{bitshift}(A, k) \]
\[ C = \text{bitshift}(A, k, n) \]

Description
\( C = \text{bitshift}(A, k) \) returns the value of \( A \) shifted by \( k \) bits. Input argument \( A \) is usually an unsigned integer. Shifting by \( k \) is the same as multiplication by \( 2^k \). Negative values of \( k \) are allowed and this corresponds to shifting to the right, or dividing by \( 2^\text{ABS}(k) \) and truncating to an integer.

If the shift causes \( C \) to overflow the number of bits in the unsigned integer class of \( A \), then the overflowing bits are dropped. If \( A \) is a double precision variable, then its value must be an integer integer between 0 and BITMAX and overflow happens after 53 bits.

\( C = \text{bitshift}(A, k, n) \) where \( A \) is double precision, causes any bits that overflow \( n \) bits to be dropped. the value of \( n \) must be less than or equal to 53.

Instead of using \( \text{bitshift}(a, k, 8) \) or another power of 2 for \( n \), consider using \( \text{bitshift}(\text{uint8}(a), k) \) or the appropriate unsigned integer class for \( A \).

Examples

Example 1
Shifting 1100 (12, decimal) to the left two bits yields 110000 (48, decimal).

\[ C = \text{bitshift}(12, 2) \]
\[ C = 48 \]

Example 2
Repeatedly shift the bits of an unsigned 16 bit value to the left until all the nonzero bits overflow. Track the progress in binary:

\[ a = \text{intmax}('\text{uint16}') \]
\[ \text{disp(sprintf(} \ldots \]
\[ '\text{Initial uint16 value } %5d \text{ is } %16s \text{ in binary}', \ldots \]
\[ a, \text{dec2bin(a)}) \]

\[ \text{for } k = 1:16 \]
\[ a = \text{bitshift}(a, 1) \]
\[ \text{disp(sprintf(} \ldots \]
'Shifted uint16 value %5d is %16s in binary',...a, dec2bin(a))
end

Repeat this experiment, this time using a double precision variable:

```
a = double(intmax('uint16'));
disp(sprintf( ...    'Initial double value %5d is %16s in binary', ...
        a, dec2bin(a)))

    for k = 1:16
        a = bitshift(a, 1, 16);
        disp(sprintf( ...    'Shifted double value %5d is %16s in binary',...
            a, dec2bin(a)))
    end
```

Now notice the difference with letting the double precision variable overflow at its default 53 bits. For brevity, shift by 3 each time:

```
a = double(intmax('uint16'));
disp(sprintf( ...    'Initial double value %16.0f is %53s in binary', ...
        a, dec2bin(a)))

    for i = 1:18
        a = bitshift(a, 3);
        disp(sprintf( ...    'Shifted double value %16.0f is %53s in binary',...
            a, dec2bin(a)))
    end
```

See Also ...
**Purpose**

Bitwise XOR

**Syntax**

```
C = bitxor(A, B)
```

**Description**

`C = bitxor(A, B)` returns the bitwise XOR of the two arguments `A` and `B`. Both `A` and `B` must be unsigned integers.

**Examples**

**Example 1**

The five-bit binary representations of the integers 13 and 27 are 01101 and 11011, respectively. Performing a bitwise XOR on these numbers yields 10110, or 22.

```
C = bitxor(uint8(13), uint8(27))
C =
22
```

**Example 2**

Create a truth table for a logical XOR operation:

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);

TT = bitxor(A, B)
TT =
    0  1
    1  0
```

**See Also**

`bitand`, `bitcmp`, `bitget`, `bitmax`, `bitor`, `bitset`, `bitshift`
**Purpose**  
A string of blanks

**Syntax**  
`blanks(n)`

**Description**  
`blanks(n)` is a string of `n` blanks.

**Examples**  
`blanks` is useful with the `display` function. For example,

```matlab
disp([‘xxx’ blanks(20) ‘yyy’])
```

displays twenty blanks between the strings ‘xxx’ and ‘yyy’.

`disp(blanks(n))'` moves the cursor down `n` lines.

**See Also**  
`clc`, `format`, `home`
**Purpose**  Construct a block diagonal matrix from input arguments

**Syntax**  
```
out = blkdiag(a,b,c,d,...)
```

**Description**  
```
out = blkdiag(a,b,c,d,...) , where a, b, c, d, ... are matrices, outputs a block diagonal matrix of the form

\[
\begin{bmatrix}
a & 0 & 0 & 0 \\
0 & b & 0 & 0 \\
0 & 0 & c & 0 \\
0 & 0 & 0 & d
\end{bmatrix}
\]
```

The input matrices do not have to be square, nor do they have to be of equal size.

**See Also**  
diag, horzcat, vertcat
Purpose
Display axes border

Syntax
box on
box off
box
box(axes_handle,...)

Description
box on displays the boundary of the current axes.
box off does not display the boundary of the current axes.
box toggles the visible state of the current axes boundary.
box(axes_handle,...) uses the axes specified by axes_handle instead of the current axes.

Algorithm
The box function sets the axes Box property to on or off.

See Also
axes, grid
“Axes Operations” for related functions
Purpose
Terminate execution of a `for` loop or `while` loop

Syntax
break

Description
`break` terminates the execution of a `for` or `while` loop. Statements in the loop that appear after the `break` statement are not executed.

In nested loops, `break` exits only from the loop in which it occurs. Control passes to the statement that follows the end of that loop.

Remarks
`break` is not defined outside a `for` or `while` loop. Use `return` in this context instead.

Examples
The example below shows a `while` loop that reads the contents of the file `fft.m` into a MATLAB character array. A `break` statement is used to exit the `while` loop when the first empty line is encountered. The resulting character array contains the M-file help for the `fft` program.

```matlab
fid = fopen('fft.m','r');
s = ' ';
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line), break, end
    s = strvcat(s,line);
end
disp(s)
```

See Also
`for`, `while`, `end`, `continue`, `return`
Purpose
Brighten or darken colormap

Syntax
```
brighten(beta)
brighten(h,beta)
newmap = brighten(beta)
newmap = brighten(cmap,beta)
```

Description
`brighten` increases or decreases the color intensities in a colormap. The modified colormap is brighter if \(0 < \beta < 1\) and darker if \(1 < \beta < 0\).

`brighten(beta)` replaces the current colormap with a brighter or darker colormap of essentially the same colors. `brighten(beta)`, followed by `brighten(\ beta)`, where \(\beta < 1\), restores the original map.

`brighten(h,\ beta)` brightens all objects that are children of the figure having the handle \(h\).

`newmap = brighten(beta)` returns a brighter or darker version of the current colormap without changing the display.

`newmap = brighten(cmap,\ beta)` returns a brighter or darker version of the colormap \(cmap\) without changing the display.

Examples
Brighten and then darken the current colormap:
```
beta = .5; brighten(beta);
beta = .5; brighten(beta);
```

Algorithm
The values in the colormap are raised to the power of gamma, where gamma is

\[
gamma = \begin{cases} 
1 - \beta, & \beta > 0 \\
\frac{1}{1+\beta}, & \beta \leq 0 
\end{cases}
\]

`brighten` has no effect on graphics objects defined with true color.

See Also
colormap, rgbplot

“Color Operations” for related functions

Altering Colormaps for more information
**Purpose**
Execute built-in function from overloaded method

**Syntax**
builtin(function, x1, ..., xn)
[y1, ..., yn] = builtin(function, x1, ..., xn)

**Description**
builtin is used in methods that overload built-in functions to execute the original built-in function. If function is a string containing the name of a built-in function, then

builtin(function, x1, ..., xn) evaluates the specified function at the given arguments x1 through xn. The function argument must be a string containing a valid function name. function cannot be a function handle.

[y1, ..., yn] = builtin(function, x1, ..., xn) returns multiple output arguments.

**Remarks**
builtin(...) is the same as feval(...) except that it calls the original built-in version of the function even if an overloaded one exists. (For this to work you must never overload builtin.)

**See Also**
feval
Purpose
Solve boundary value problems (BVPs) for ordinary differential equations

Syntax
sol = bvp4c(odefun,bcfun,solinit)
sol = bvp4c(odefun,bcfun,solinit,options)
sol = bvp4c(odefun,bcfun,solinit,options,p1,p2...)

Arguments
odefun
A function that evaluates the differential equations \( f(x,y) \). It can have the form
\[
dydx = odefun(x,y)
dydx = odefun(x,y,p1,p2,...)
dydx = odefun(x,y,parameters)
dydx = odefun(x,y,parameters,p1,p2,...)
\]
where \( x \) is a scalar corresponding to \( x \), and \( y \) is a column vector corresponding to \( y \). \( parameters \) is a vector of unknown parameters, and \( p1,p2,... \) are known parameters. The output \( dydx \) is a column vector.

bcfun
A function that computes the residual in the boundary conditions. For two-point boundary value conditions of the form \( bc(y(a),y(b)) \), \( bcfun \) can have the form
\[
res = bcfun(ya,yb)
res = bcfun(ya,yb,p1,p2,...)
res = bcfun(ya,yb,parameters)
res = bcfun(ya,yb,parameters,p1,p2,...)
\]
where \( ya \) and \( yb \) are column vectors corresponding to \( y(a) \) and \( y(b) \). \( parameters \) is a vector of unknown parameters, and \( p1,p2,... \) are known parameters. The output \( res \) is a column vector.

See “Multipoint Boundary Value Problems” on page 2-258 for a description of \( bcfun \) for multipoint boundary value problems.

solinit
A structure containing the initial guess for a solution. You create \( solinit \) using the function \( bvpinit \). \( solinit \) has the following fields.

\( x \)
Ordered nodes of the initial mesh. Boundary conditions are imposed at \( a = solinit.x(1) \) and \( b = solinit.x(end) \). 

2-255
bvp4c

sol = bvp4c(odefun,bcfun,solinit)

integrates a system of ordinary differential equations of the form

\[ y' = f(x,y) \]

on the interval \( [a,b] \) subject to two-point boundary value conditions

\[ bc(y(a),y(b)) = 0 \]

bvp4c can also solve multipoint boundary value problems. See “Multipoint Boundary Value Problems” on page 2-258. You can use the function bvpinit to specify the boundary points, which are stored in the input argument solinit. See the reference page for bvpinit for more information.

The bvp4c solver can also find unknown parameters \( p \) for problems of the form

\[ y' = f(x,y,p) \]
\[ 0 = bc(y(a),y(b),p) \]

where \( p \) corresponds to parameters. You provide bvp4c an initial guess for any unknown parameters in solinit.parameters. The bvp4c solver returns the final values of these unknown parameters in sol.parameters.

bvp4c produces a solution that is continuous on \( [a,b] \) and has a continuous first derivative there. Use the function deval and the output sol of bvp4c to evaluate the solution at specific points xint in the interval \( [a,b] \).
sxint = deval(sol,xint)

The structure sol returned by bvp4c has the following fields:

- **sol.x**: Mesh selected by bvp4c
- **sol.y**: Approximation to $y(x)$ at the mesh points of sol.x
- **sol.yp**: Approximation to $y'(x)$ at the mesh points of sol.x
- **sol.parameters**: Values returned by bvp4c for the unknown parameters, if any
- **sol.solver**: 'bvp4c'

The structure sol can have any name, and bvp4c creates the fields x, y, yp, parameters, and solver.

- **sol = bvp4c(odefun,bcfun,solinit,options)** solves as above with default integration properties replaced by the values in options, a structure created with the bvpset function. See bvpset for details.

- **sol = bvp4c(odefun,bcfun,solinit,options,p1,p2...)** passes constant known parameters, $p_1, p_2, ...$, to odefun, bcfun, and all the functions the user specifies in options. Use options = [] as a placeholder if no options are set.

    at any point in $[a,b]$. If there are unknown parameters,

- **solinit = bvpinit(x, yinit, params)** forms the initial guess solinit with the vector params of guesses for the unknown parameters.

**Singular Boundary Value Problems**

bvp4c solves a class of singular boundary value problems, including problems with unknown parameters $p$, of the form

$$
y' = S \cdot y/x + f(x, y, p)\\
0 = bc(y(0), y(b), p)
$$

The interval is required to be $[0, b]$ with $b > 0$. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix $S$ as the value of the 'SingularTerm' option of bvpset, and odefun evaluates only $f(x, y, p)$. The
boundary conditions must be consistent with the necessary condition 
\[ S \cdot y(0) = 0 \] and the initial guess should satisfy this condition.

**Multipoint Boundary Value Problems**
bvp4c can solve multipoint boundary value problems where 
\[ a = a_0 < a_1 < a_2 < \ldots < a_n = b \] are boundary points in the interval \( [a, b] \). The 
points \( a_1, a_2, \ldots, a_{n-1} \) represent interfaces that divide \( [a, b] \) into regions. 
bvp4c enumerates the regions from left to right (from \( a \) to \( b \)), with indices 
starting from 1. In region \( k \), \([a_{k-1}, a_k] \), bvp4c evaluates the derivative as 
\[ yp = odefun(x, y, k) \]

In the boundary conditions function 
\[ \text{bcfun}(yleft, yright) \]
\( yleft(:, k) \) is the solution at the left boundary of \([a_{k-1}, a_k]\). Similarly, 
\( yright(:, k) \) is the solution at the right boundary of region \( k \). In particular, 
\[ yleft(:, 1) = y(a) \]
and 
\[ yright(:, end) = y(b) \]

For example, if there just one equation and the boundary points are \( 0 < 1 < 2 \), 
to specify the boundary conditions 

\[ y(0) = 4, y(1) = 4.5 \text{ on } [0,1] \]

\[ y(1) = 5, y(1) = 5.5 \text{ on } [1,2] \]

\( yleft \) and \( yright \) have the following values.
\[ \text{yleft} = [4; 5]; \]
\[ \text{yright} = [4.5; 5.5]; \]

The boundary condition function \( \text{bcfun} \) has the form
\[ \text{function res = bc(yleft, yright)} \]
\[ \text{res} = [ yleft(1) - 4 } \]
\[ \text{yleft(1) - 4.5} \]
\[ \text{yleft(2) - 5 } \]
\[ \text{yright(2) - 5.5} ]; \]
When you create an initial guess with

\[
\text{solinit} = \text{bvpinit(xinit, yinit)},
\]

use double entries in \text{xinit} for each interface point. See the reference page for \text{bvpinit} for more information.

If \text{yinit} is a function, \text{bvpinit} calls \( y = \text{yinit}(x, k) \) to get an initial guess for the solution at \( x \) in region \( k \). In the solution structure \text{sol} returned by \text{bvp4c}, \text{sol.x} has double entries for each interface point. The corresponding columns of \text{sol.y} contain the left and right solution at the interface, respectively.

For an example of solving a three-point boundary value problem, enter

```
threebvp
```

**Examples**

**Example 1.** Boundary value problems can have multiple solutions and one purpose of the initial guess is to indicate which solution you want. The second order differential equation

\[
y'' + |y| = 0
\]

has exactly two solutions that satisfy the boundary conditions

\[
y(0) = 0
y(4) = -2
\]

Prior to solving this problem with \text{bvp4c}, you must write the differential equation as a system of two first order ODEs

\[
\begin{align*}
y_1' &= y_2 \\
y_2' &= -|y_1|
\end{align*}
\]

Here \( y_1 = y \) and \( y_2 = y' \). This system has the required form

\[
\begin{align*}
y' &= f(x, y) \\
bc(y(a), y(b)) &= 0
\end{align*}
\]

The function \( f \) and the boundary conditions \( bc \) are coded in MATLAB as functions \text{twoode} and \text{twobc}.
function dydx = twoode(x,y)
  dydx = [ y(2)
           -abs(y(1))];

function res = twobc(ya,yb)
  res = [ ya(1)
          yb(1) + 2];

Form a guess structure consisting of an initial mesh of five equally spaced
points in [0,4] and a guess of constant values $y_1(x) = 1$ and $y_2(x) = 0$ with the
command

solinit = bvpinit(linspace(0,4,5),[1 0]);

Now solve the problem with

sol = bvp4c(@twoode,@twobc,solinit);

Evaluate the numerical solution at 100 equally spaced points and plot $y(x)$
with

x = linspace(0,4);
y = deval(sol,x);
plot(x,y(1,:));
You can obtain the other solution of this problem with the initial guess

\[
solinit = \text{bvpinit}(\text{linspace}(0,4,5),[-1 0]);
\]

**Example 2.** This boundary value problem involves an unknown parameter. The task is to compute the fourth \((q = 5)\) eigenvalue \(\lambda\) of Mathieu's equation

\[
y'' + (\lambda - 2q \cos 2x)y = 0
\]

Because the unknown parameter \(\lambda\) is present, this second order differential equation is subject to *three* boundary conditions

\[
\begin{align*}
y'(0) &= 0 \\
y'(\pi) &= 0 \\
y(0) &= 1
\end{align*}
\]

It is convenient to use subfunctions to place all the functions required by bvp4c in a single M-file.

```matlab
function mat4bvp

lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
sol = bvp4c(@mat4ode,@mat4bc,solinit);
```

2-261
fprintf('The fourth eigenvalue is approximately %7.3f.
',
sol.parameters)

xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
axis([0 pi -1 1.1])
title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')

function dydx = mat4ode(x,y,lambda)
q = 5;
dydx = [ y(2)
         -(lambda - 2*q*cos(2*x))*y(1) ];

function res = mat4bc(ya,yb,lambda)
res = [ ya(2)
        yb(2)
        ya(1)-1 ];

function yinit = mat4init(x)
yinit = [ cos(4*x)
         -4*sin(4*x) ];

The differential equation (converted to a first order system) and the boundary
conditions are coded as subfunctions mat4ode and mat4bc, respectively.
Because unknown parameters are present, these functions must accept three
input arguments, even though some of the arguments are not used.

The guess structure solinit is formed with bvpinit. An initial guess for the
solution is supplied in the form of a function mat4init. We chose \( y = \cos 4x \)
because it satisfies the boundary conditions and has the correct qualitative
behavior (the correct number of sign changes). In the call to bvpinit, the third
argument (\( \text{lambda} = 15 \)) provides an initial guess for the unknown parameter \( \lambda \).
After the problem is solved with \texttt{bvp4c}, the field \texttt{sol.parameters} returns the value $\lambda = 17.097$, and the plot shows the eigenfunction associated with this eigenvalue.

![Eigenfunction of Mathieu's equation.](image)

**Algorithms**
\texttt{bvp4c} is a finite difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a $C^1$-continuous solution that is fourth order accurate uniformly in $[a,b]$. Mesh selection and error control are based on the residual of the continuous solution.

**See Also**
@\texttt{(function\_handle)}, \texttt{bvpget}, \texttt{bvpinit}, \texttt{bvpset}, \texttt{deval}

**References**

2-263
bvpget

Purpose
Extract properties from the options structure created with bvpset

Syntax
val = bvpget(options,'name')
val = bvpget(options,'name',default)

Description
val = bvpget(options,'name') extracts the value of the named property from the structure options, returning an empty matrix if the property value is not specified in options. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. [ ] is a valid options argument.

val = bvpget(options,'name',default) extracts the named property as above, but returns val = default if the named property is not specified in options. For example,

val = bvpget(opts,'RelTol',1e-4);

returns val = 1e-4 if the RelTol is not specified in opts.

See Also
bvp4c, bvpinit, bvpset, deval
bvpinit

**Purpose**

Form the initial guess for bvp4c

**Syntax**

solinit = bvpinit(x,yinit)
solinit = bvpinit(x,yinit,parameters)
solinit = bvpinit(sol,[anew bnew])
solinit = bvpinit(sol,[anew bnew],parameters)

**Description**

solinit = bvpinit(x,yinit) forms the initial guess for the boundary value problem solver bvp4c.

x is a vector that specifies an initial mesh. If you want to solve the boundary value problem (BVP) on \([a, b]\), then specify \(x(1)\) as \(a\) and \(x(\text{end})\) as \(b\). The function bvp4c adapts this mesh to the solution, so a guess like \(x = \text{linspace}(a,b,10)\) often suffices. However, in difficult cases, you should place mesh points where the solution changes rapidly. The entries of \(x\) must be in

- Increasing order if \(a < b\)
- Decreasing order if \(a > b\)

For two-point boundary value problems, the entries of \(x\) must be distinct. That is, if \(a < b\), the entries must satisfy \(x(1) < x(2) < ... < x(\text{end})\). If \(a > b\), the entries must satisfy \(x(1) > x(2) > ... > x(\text{end})\).

For multipoint boundary value problem, you can specify the points in \([a, b]\) at which the boundary conditions apply, other than the endpoints \(a\) and \(b\), by repeating their entries in \(x\). For example, if you set

\[
x = [0, 0.5, 1, 1, 1.5, 2];
\]

the boundary conditions apply at three points: the endpoints 0 and 2, and the repeated entry 1. In general, repeated entries represent boundary points between regions in \([a, b]\). In the preceding example, the repeated entry 1 divides the interval \([0,2]\) into two regions: \([0,1]\) and \([1,2]\).

yinit is a guess for the solution. It can be either a vector, or a function:

- **Vector** – For each component of the solution, bvpinit replicates the corresponding element of the vector as a constant guess across all mesh points. That is, \(yinit(i)\) is a constant guess for the \(i\)th component \(yinit(i,:)\) of the solution at all the mesh points in \(x\).
Function – For a given mesh point, the guess function must return a vector whose elements are guesses for the corresponding components of the solution. The function must be of the form

\[ y = \text{guess}(x) \]

where \( x \) is a mesh point and \( y \) is a vector whose length is the same as the number of components in the solution. For example, if the guess function is an M-file function, \texttt{bvpinit} calls

\[ y(:,j) = \text{@guess}(x(j)) \]

at each mesh point.

For multipoint boundary value problems, the guess function must be of the form

\[ y = \text{guess}(x, k) \]

where \( y \) is an initial guess for the solution at \( x \) in region \( k \). The function must accept the input argument \( k \), which is provided for flexibility in writing the guess function. However, the function is not required to use \( k \).

\texttt{solinit = bvpinit(x, yinit, parameters)} indicates that the boundary value problem involves unknown parameters. Use the vector \texttt{parameters} to provide a guess for all unknown parameters.

\texttt{solinit} is a structure with the following fields. The structure can have any name, but the fields must be named \( x \), \( y \), and \( \text{parameters} \).

\begin{itemize}
  \item \texttt{x} \quad \text{Ordered nodes of the initial mesh.}
  \item \texttt{y} \quad \text{Initial guess for the solution with solinit.y(:,i) a guess for the solution at the node solinit.x(i).}
  \item \texttt{parameters} \quad \text{Optional. A vector that provides an initial guess for unknown parameters.}
\end{itemize}

\texttt{solinit = bvpinit(x, yinit, parameters, p1, p2...)} passes the additional known parameters \( p_1, p_2, ... \) to the guess function \texttt{yinit} as

\[ \text{yinit}(x, p_1, p_2...) \]

for two-point boundary value problems, or as

\[ \text{yinit}(x, k, p_1, p_2) \]

for multipoint boundary value problems. You can only use known parameters \( p_1, p_2, ... \) when \texttt{yinit} is a function. When there are no unknown parameters, pass in \([\ ]\) for parameters.
solinit = bvpinit(sol, [anew bnew]) forms an initial guess on the interval [anew bnew] from a solution sol on an interval [a, b]. The new interval must be larger than the previous one, so either anew <= a < b <= bnew or anew >= a > b >= bnew. The solution sol is extrapolated to the new interval. If sol contains parameters, they are copied to solinit.

solinit = bvpinit(sol, [anew bnew], parameters) forms solinit as described above, but uses parameters as a guess for unknown parameters in solinit.

See Also

@(function_handle), bvp4c, bvpget, bvpset, deval
bvpset

Purpose
Create/alter boundary value problem (BVP) options structure

Syntax
options = bvpset('name1',value1,'name2',value2,...)
options = bvpset(oldopts 'name1',value1,...)
options = bvpset(oldopts,newopts)
bvpset

Description
options = bvpset('name1',value1,'name2',value2,...) creates a structure options in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names.

options = bvpset(oldopts,'name1',value1,...) alters an existing options structure oldopts.

options = bvpset(oldopts,newopts) combines an existing options structure oldopts with a new options structure newopts. Any new properties overwrite corresponding old properties.

bvpset with no input arguments displays all property names and their possible values.

BVP Properties
These properties are available.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelTol</td>
<td>Positive scalar {1e-3}</td>
<td>A relative tolerance that applies to all components of the residual vector. The computed solution $S(x)$ is the exact solution of $S'(x) = F(x,S(x)) + res(x)$. On each subinterval of the mesh, the residual $res(x)$ satisfies $|res(i)/\text{max}(abs(F(i)),\text{AbsTol}(i)/\text{RelTol})| \leq \text{RelTol}$</td>
</tr>
<tr>
<td>AbsTol</td>
<td>Positive scalar or vector [1e-6]</td>
<td>An absolute tolerance that applies to all components of the residual vector. Elements of a vector of tolerances apply to corresponding components of the residual vector.</td>
</tr>
<tr>
<td>Property</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Vectorized</td>
<td>on</td>
<td>[off]</td>
</tr>
<tr>
<td>SingularTerm</td>
<td>Matrix</td>
<td>Singular term of singular BVPs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set to the constant matrix $S$ for equations of the form $y' = S\frac{y}{x} + f(x,y,p)$ that are posed on the interval $[0, b]$ where $b &gt; 0$.</td>
</tr>
<tr>
<td>FJacobian</td>
<td>Function</td>
<td>matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, when solving $y' = f(x,y)$, set this property to $@FJAC$ if $DFDY = FJAC(X,Y)$ evaluates the Jacobian of $f$ with respect to $y$. If the problem involves unknown parameters $p$, $[DFDY,DFDP] = FJAC(X,Y,P)$ must also return the partial derivative of $f$ with respect to $p$. For problems with constant partial derivatives, set this property to the value of $DFDY$ or to a cell array ${DFDY,DFDP}$.</td>
</tr>
<tr>
<td>BCJacobian</td>
<td>Function</td>
<td>cell array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, for boundary conditions $bc(ya,yb) = 0$, set this property to $@BCJAC$ if $[DBCYA,DBCYB] = BCJAC(YA,YB)$ evaluates the partial derivatives of $bc$ with respect to $ya$ and to $yb$. If the problem involves unknown parameters $p$, then $[DBCYA,DBCYB,DBCDP] = BCJAC(YA,YB,P)$ must also return the partial derivative of $bc$ with respect to $p$. For problems with constant partial derivatives, set this property to a cell array ${DBCYA,DBCYB}$ or ${DBCYA,DBCYB,DBCDP}$.</td>
</tr>
</tbody>
</table>
# bvpset

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nmax</td>
<td>positive integer</td>
<td>Maximum number of mesh points allowed.</td>
</tr>
<tr>
<td></td>
<td>{floor(1000/n)}</td>
<td></td>
</tr>
<tr>
<td>Stats</td>
<td>on {off}</td>
<td>Display computational cost statistics.</td>
</tr>
</tbody>
</table>

## See Also

@{(function_handle), bvp4c, bvpget, bvpinit, deval}
Purpose

Calendar

Syntax

c = calendar

c = calendar(d)

c = calendar(y,m)

calendar(...)

Description

c = calendar returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.

c = calendar(d), where d is a serial date number or a date string, returns a calendar for the specified month.

c = calendar(y,m), where y and m are integers, returns a calendar for the specified month of the specified year.

calendar(...) displays the calendar on the screen.

Examples

The command

calendar(1957,10)

reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

See Also

datenum
Purpose
Move the camera position and target

Syntax
- camdolly(dx,dy,dz)
- camdolly(dx,dy,dz,'targetmode')
- camdolly(dx,dy,dz,'targetmode','coordsys')
- camdolly(axes_handle,...)

Description
camdolly moves the camera position and the camera target by the specified amounts.

camdolly(dx,dy,dz) moves the camera position and the camera target by the specified amounts (see “Coordinate Systems”).

camdolly(dx,dy,dz,'targetmode') The targetmode argument can take on two values that determine how MATLAB moves the camera:

- movetarget (default) — Move both the camera and the target.
- fixtarget — Move only the camera.

ncamdolly(dx,dy,dz,'targetmode','coordsys') The coordsys argument can take on three values that determine how MATLAB interprets dx, dy, and dz:

Coordinate Systems

- camera (default) — Move in the camera’s coordinate system. dx moves left/right, dy moves down/up, and dz moves along the viewing axis. The units are normalized to the scene.
  For example, setting dx to 1 moves the camera to the right, which pushes the scene to the left edge of the box formed by the axes position rectangle. A negative value moves the scene in the other direction. Setting dz to 0.5 moves the camera to a position halfway between the camera position and the camera target.
- pixels — Interpret dx and dy as pixel offsets. dz is ignored.
- data — Interpret dx, dy, and dz as offsets in axes data coordinates.

camdolly(axes_handle,...) operates on the axes identified by the first argument, axes_handle. When you do not specify an axes handle, camdolly operates on the current axes.
Remarks

camdolly sets the axes CameraPosition and CameraTarget properties, which in turn causes the CameraPositionMode and CameraTargetMode properties to be set to manual.

Examples

This example moves the camera along the $x$- and $y$-axes in a series of steps.

```
surf(peaks)
axis vis3d
t = 0:pi/20:2*pi;
dx = sin(t)./40;
dy = cos(t)./40;
for i = 1:length(t);
    camdolly(dx(i),dy(i),0)
    drawnow
end
```

See Also

axes, campos, camproj, camtarget, camup, camva

The axes properties CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection

“Controlling the Camera Viewpoint” for related functions

See Defining Scenes with Camera Graphics for more information on camera properties.
**cameratoolbar**

**Purpose**
Control camera toolbar programmatically

**Syntax**
cameratoolbar
cameratoolbar('NoReset')
cameratoolbar('SetMode', mode)
cameratoolbar('SetCoordSys', coordsys)
cameratoolbar('Show')
cameratoolbar('Hide')
cameratoolbar('Toggle')
cameratoolbar('ResetCameraAndSceneLight')
cameratoolbar('ResetCamera')
cameratoolbar('ResetSceneLight')
cameratoolbar('ResetTarget')
mode = cameratoolbar('GetMode')
paxis = cameratoolbar('GetCoordsys')
vis = cameratoolbar('GetVisible')
h = cameratoolbar
 cameratoolbar('Close')

**Description**
cameratoolbar creates a new toolbar that enables interactive manipulation of the axes camera and light when users drag the mouse on the figure window. Several axes camera properties are set when the toolbar is initialized.

cameratoolbar('NoReset') creates the toolbar without setting any camera properties.

cameratoolbar('SetMode', mode) sets the toolbar mode (depressed button). mode can be: 'orbit', 'orbitscenelight', 'pan', 'dollyhv', 'dollyfb', 'zoom', 'roll', 'nomode'.

cameratoolbar('SetCoordSys', coordsys) sets the principal axis of the camera motion. coordsys can be: 'x', 'y', 'z', 'none'.

cameratoolbar('Show') shows the toolbar on the current figure.

cameratoolbar('Hide') hides the toolbar on the current figure.

cameratoolbar('Toggle') toggles the visibility of the toolbar.
cameratoolbar('ResetCameraAndSceneLight') resets the current camera and scenelight.

cameratoolbar('ResetCamera') resets the current camera.

cameratoolbar('ResetSceneLight') resets the current scenelight.

cameratoolbar('ResetTarget') resets the current camera target.

mode = cameratoolbar('GetMode') returns the current mode.

paxis = cameratoolbar('GetCoordsys') returns the current principal axis.

vis = cameratoolbar('GetVisible') returns the visibility of the toolbar (1 if visible, 0 if not visible).

h = cameratoolbar returns the handle to the toolbar.

cameratoolbar('Close') removes the toolbar from the current figure.

Note that, in general, the use of OpenGL hardware improves rendering performance.

See Also

rotate3d, zoom
camlight

**Purpose**
Create or move a light object in camera coordinates

**Syntax**
camlight headlight
    camlight right
    camlight left
    camlight
    camlight(az,el)
    camlight(...,'style')
    camlight(light_handle,...)
    light_handle = camlight(...)

**Description**
camlight('headlight') creates a light at the camera position.
camlight('right') creates a light right and up from camera.
camlight('left') creates a light left and up from camera.
camlight with no arguments is the same as camlight('right').
camlight(az,el) creates a light at the specified azimuth (az) and elevation (el) with respect to the camera position. The camera target is the center of rotation and az and el are in degrees.
camlight(...,'style') The style argument can take on two values:

- local (default) — The light is a point source that radiates from the location in all directions.
- infinite — The light shines in parallel rays.
camlight(light_handle,...) uses the light specified in light_handle.
    light_handle = camlight(...) returns the light's handle.

**Remarks**
camlight sets the light object Position and Style properties. A light created with camlight will not track the camera. In order for the light to stay in a constant position relative to the camera, you must call camlight whenever you move the camera.
Examples

This example creates a light positioned to the left of the camera and then repositions the light each time the camera is moved:

```matlab
surf(peaks)
axis vis3d
h = camlight('left');
for i = 1:20;
    camorbit(10,0)
    camlight(h,'left')
    drawnow;
end
```

See Also

light, lightangle

“Lighting” for related functions

Lighting as a Visualization Tool for more information on using lights
camlookat

Purpose
Position the camera to view an object or group of objects.

Syntax
- `camlookat(object_handles)`
- `camlookat(axes_handle)`
- `camlookat`

Description
- `camlookat(object_handles)` views the objects identified in the vector `object_handles`. The vector can contain the handles of axes children.

- `camlookat(axes_handle)` views the objects that are children of the axes identified by `axes_handle`.

- `camlookat` views the objects that are in the current axes.

Remarks
`camlookat` moves the camera position and camera target while preserving the relative view direction and camera view angle. The object (or objects) being viewed roughly fill the axes position rectangle.

`camlookat` sets the axes `CameraPosition` and `CameraTarget` properties.

Examples
This example creates three spheres at different locations and then progressively positions the camera so that each sphere is the object around which the scene is composed:

```matlab
[x y z] = sphere;
s1 = surf(x,y,z);
hold on
s2 = surf(x+3,y,z+3);
s3 = surf(x,y,z+6);
daspect([1 1 1])
view(30,10)
camproj perspective
camlookat(gca) % Compose the scene around the current axes
pause(2)
camlookat(s1) % Compose the scene around sphere s1
pause(2)
camlookat(s2) % Compose the scene around sphere s2
pause(2)
camlookat(s3) % Compose the scene around sphere s3
pause(2)
camlookat(gca)
```
See Also  campos, camtarget

“Controlling the Camera Viewpoint” for related functions

Defining Scenes with Camera Graphics for more information
### Purpose
Rotate the camera position around the camera target

### Syntax
```matlab
camorbit(dtheta,dphi)
camorbit(dtheta,dphi,'coordsys')
camorbit(dtheta,dphi,'coordsys','direction')
camorbit(axes_handle,...)
```

### Description
`camorbit(dtheta,dphi)` rotates the camera position around the camera target by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`camorbit(dtheta,dphi,'coordsys')` The `coordsys` argument determines the center of rotation. It can take on two values:

- **data** (default) — Rotate the camera around an axis defined by the camera target and the direction (default is the positive z direction).
- **camera** — Rotate the camera about the point defined by the camera target.

`camorbit(dtheta,dphi,'coordsys','direction')` The `direction` argument, in conjunction with the camera target, defines the axis of rotation for the data coordinate system. Specify `direction` as a three-element vector containing the x, y, and z components of the direction or one of the characters, `x`, `y`, or `z`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`camorbit(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camorbit` operates on the current axes.

### Examples
Compare rotation in the two coordinate systems with these for loops. The first rotates the camera horizontally about a line defined by the camera target point and a direction that is parallel to the y-axis. Visualize this rotation as a cone formed with the camera target at the apex and the camera position forming the base:

```matlab
surf(peaks)
axis vis3d
for i=1:36
    camorbit(10,0,'data',[0 1 0])
drawnow
```
Rotation in the camera coordinate system orbits the camera around the axes along a circle while keeping the center of a circle at the camera target.

```matlab
surf(peaks)
axis vis3d
for i=1:36
    camorbit(10,0,'camera')
    drawnow
end
```

**See Also**

axes, axis('vis3d'), camdolly, campan, camzoom, camroll

“Controlling the Camera Viewpoint” for related functions

Defining Scenes with Camera Graphics for more information
campan

**Purpose**

Rotate the camera target around the camera position

**Syntax**

```matlab
campan(dtheta,dphi)
campan(dtheta,dphi,'coordsys')
campan(dtheta,dphi,'coordsys','direction')
campan(axes_handle,...)
```

**Description**

`campan(dtheta,dphi)` rotates the camera target around the camera position by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`campan(dtheta,dphi,'coordsys')` The `coordsys` argument determines the center of rotation. It can take on two values:

- data (default) — Rotate the camera target around an axis defined by the camera position and the direction (default is the positive z direction)
- camera — Rotate the camera about the point defined by the camera target.

`campan(dtheta,dphi,'coordsys','direction')` The `direction` argument, in conjunction with the camera position, defines the axis of rotation for the data coordinate system. Specify direction as a three-element vector containing the x, y, and z components of the direction or one of the characters, x, y, or z, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`campan(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `campan` operates on the current axes.

**See Also**

`axes`, `camdolly`, `camorbit`, `camtarget`, `camzoom`, `camroll`

“Controlling the Camera Viewpoint” for related functions

Defining Scenes with Camera Graphics for more information
**Purpose**
Set or query the camera position

**Syntax**
campos
campos([camera_position])
campos('mode')
campos('auto')
campos('manual')
campos(axes_handle,...)

**Description**
campos with no arguments returns the camera position in the current axes.
campos([camera_position]) sets the position of the camera in the current axes to the specified value. Specify the position as a three-element vector containing the x-, y-, and z-coordinates of the desired location in the data units of the axes.
campos('mode') returns the value of the camera position mode, which can be either auto (the default) or manual.
campos('auto') sets the camera position mode to auto.
campos('manual') sets the camera position mode to manual.
campos(axes_handle,...) performs the set or query on the axes identified by the first argument, axes_handle. When you do not specify an axes handle, campos operates on the current axes.

**Remarks**
campos sets or queries values of the axes CameraPosition and CameraPositionMode properties. The camera position is the point in the Cartesian coordinate system of the axes from which you view the scene.

**Examples**
This example moves the camera along the x-axis in a series of steps:
surf(peaks)
axis vis3d off
for x = −200:5:200
    campos([x,5,10])
drawnow
end
See Also

axis, camproj, camtarget, camup, camva

The axes properties CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection

“Controlling the Camera Viewpoint” for related functions

Defining Scenes with Camera Graphics for more information
Purpose
Set or query the projection type

Syntax
```
camproj

camproj(projection_type)
camproj(axes_handle,...)
```

Description
The projection type determines whether MATLAB uses a perspective or orthographic projection for 3-D views.

```
camproj with no arguments returns the projection type setting in the current axes.

camproj('projection_type') sets the projection type in the current axes to the specified value. Possible values for projection_type are orthographic and perspective.

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

Remarks
camproj sets or queries values of the axes object Projection property.

See Also
campos, camtarget, camup, camva
The axes properties CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection
“Controlling the Camera Viewpoint” for related functions
Defining Scenes with Camera Graphics for more information
**camroll**

**Purpose**
Rotate the camera about the view axis

**Syntax**
camroll(dtheta)
camroll(axes_handle, dtheta)

**Description**
camroll(dtheta) rotates the camera around the camera viewing axis by the amounts specified in dtheta (in degrees). The viewing axis is defined by the line passing through the camera position and the camera target.

CAMROLL(AXES_HANDLE, DTHETA) operates on the axes identified by the first argument, axes_handle. When you do not specify an axes handle, CAMROLL operates on the current axes.

**Remarks**
camroll sets the axes CameraUpVector property and thereby also sets the CameraUpVectorMode property to manual.

**See Also**
axes, axis('vis3d'), camdolly, camorbit, camzoom, campan
“Controlling the Camera Viewpoint” for related functions
Defining Scenes with Camera Graphics for more information
Purpose
Set or query the location of the camera target

Syntax
camtarget

- camtarget([camera_target])
- camtarget('mode')
- camtarget('auto')
- camtarget('manual')
- camtarget(axes_handle,...)

Description
The camera target is the location in the axes that the camera points to. The camera remains oriented toward this point regardless of its position.

camtarget with no arguments returns the location of the camera target in the current axes.

camtarget([camera_target]) sets the camera target in the current axes to the specified value. Specify the target as a three-element vector containing the x-, y-, and z-coordinates of the desired location in the data units of the axes.

camtarget('mode') returns the value of the camera target mode, which can be either auto (the default) or manual.

camtarget('auto') sets the camera target mode to auto.

- camtarget('manual') sets the camera target mode to manual.

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

Remarks
camtarget sets or queries values of the axes object CameraTarget and CameraTargetMode properties.

- When the camera target mode is auto, MATLAB positions the camera target at the center of the axes plot box.

Examples
This example moves the camera position and the camera target along the x-axis in a series of steps:

- surf(peaks);
axis vis3d
xp = linspace(-150, 40, 50);
xt = linspace(25, 50, 50);
for i=1:50
    campos([xp(i), 25, 5]);
    camtarget([xt(i), 30, 0])
    drawnow
end

See Also
axis, camproj, campos, camup, camva

The axes properties CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection

“Controlling the Camera Viewpoint” for related functions
Defining Scenes with Camera Graphics for more information
Purpose
Set or query the camera up vector

Syntax
camup
camup([up_vector])
camup('mode')
camup('auto')
camup('manual')
camup(axes_handle,...)

Description
The camera up vector specifies the direction that is oriented up in the scene.
camup with no arguments returns the camera up vector setting in the current axes.
camup([up_vector]) sets the up vector in the current axes to the specified value. Specify the up vector as x, y, and z components. See Remarks.
camup('mode') returns the current value of the camera up vector mode, which can be either auto (the default) or manual.
camup('auto') sets the camera up vector mode to auto. In auto mode, MATLAB uses a value for the up vector of [0 1 0] for 2-D views. This means the z-axis points up.
camup('manual') sets the camera up vector mode to manual. In manual mode, MATLAB does not change the value of the camera up vector.
camup(axes_handle,...) performs the set or query on the axes identified by the first argument, axes_handle. When you do not specify an axes handle, camup operates on the current axes.

Remarks
camup sets or queries values of the axes object CameraUpVector and CameraUpVectorMode properties.
Specify the camera up vector as the x-, y-, and z-coordinates of a point in the axes coordinate system that forms the directed line segment PQ, where P is the point (0,0,0) and Q is the specified x-, y-, and z-coordinates. This line always points up. The length of the line PQ has no effect on the orientation of the scene. This means a value of [0 0 1] produces the same results as [0 0 25].
See Also

axis, camproj, campos, camtarget, camva

The axes properties CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection

“Controlling the Camera Viewpoint” for related functions

Defining Scenes with Camera Graphics for more information
Purpose
Set or query the camera view angle

Syntax
```
camva
camva(view_angle)
camva('mode')
camva('auto')
camva('manual')
camva(axes_handle,...)
```

Description
The camera view angle determines the field of view of the camera. Larger angles produce a smaller view of the scene. You can implement zooming by changing the camera view angle.

`camva` with no arguments returns the camera view angle setting in the current axes.

`camva(view_angle)` sets the view angle in the current axes to the specified value. Specify the view angle in degrees.

`camva('mode')` returns the current value of the camera view angle mode, which can be either `auto` (the default) or `manual`. See Remarks.

`camva('auto')` sets the camera view angle mode to `auto`.

`camva('manual')` sets the camera view angle mode to `manual`. See Remarks.

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

Remarks
`camva` sets or queries values of the axes object `CameraViewAngle` and `CameraViewAngleMode` properties.

When the camera view angle mode is `auto`, MATLAB adjusts the camera view angle so that the scene fills the available space in the window. If you move the camera to a different position, MATLAB changes the camera view angle to maintain a view of the scene that fills the available area in the window.
Setting a camera view angle or setting the camera view angle to manual disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the camera view angle to its current value, 

\[
camva(camva)
\]
can cause a change in the way the graph looks. See the Remarks section of the \texttt{axes} reference page for more information.

\textbf{Examples}

This example creates two pushbuttons, one that zooms in and another that zooms out.

```matlab
uicontrol('Style','pushbutton',...
     'String','Zoom In',...
     'Position',[20 20 60 20],...
     'Callback','if camva <= 1;return;else;camva(camva-1);end');
uicontrol('Style','pushbutton',...
     'String','Zoom Out',...
     'Position',[100 20 60 20],...
     'Callback','if camva >= 179;return;else;camva(camva+1);end');
```

Now create a graph to zoom in and out on:

```matlab
surf(peaks);
```

Note the range checking in the callback statements. This keeps the values for the camera view angle in the range greater than zero and less than 180.

\textbf{See Also}

\texttt{axis, camproj, campos, camup, camtarget}

The axes properties \texttt{CameraPosition, CameraTarget, CameraUpVector, CameraViewAngle, Projection}

“Controlling the Camera Viewpoint” for related functions

Defining Scenes with Camera Graphics for more information
Purpose
Zoom in and out on a scene

Syntax
camzoom(zoom_factor)
camzoom(axes_handle,...)

Description
camzoom(zoom_factor) zooms in or out on the scene depending on the value specified by zoom_factor. If zoom_factor is greater than 1, the scene appears larger; if zoom_factor is greater than zero and less than 1, the scene appears smaller.

camzoom(axes_handle,...) operates on the axes identified by the first argument, axes_handle. When you do not specify an axes handle, camzoom operates on the current axes.

Remarks
camzoom sets the axes CameraViewAngle property, which in turn causes the CameraViewAngleMode property to be set to manual. Note that setting the CameraViewAngle property disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This may result in a change to the aspect ratio of your graph. See the axes function for more information on this behavior.

See Also
axes, camdolly, camorbit, campan, camroll, camva

“Controlling the Camera Viewpoint” for related functions
Defining Scenes with Camera Graphics for more information
**Purpose**
Transform Cartesian coordinates to polar or cylindrical

**Syntax**

\[
[\text{THETA}, \text{RHO}, Z] = \text{cart2pol}(X, Y, Z) \\
[\text{THETA}, \text{RHO}] = \text{cart2pol}(X, Y)
\]

**Description**

\([\text{THETA}, \text{RHO}, Z] = \text{cart2pol}(X, Y, Z)\) transforms three-dimensional Cartesian coordinates stored in corresponding elements of arrays \(X\), \(Y\), and \(Z\), into cylindrical coordinates. \(\text{THETA}\) is a counterclockwise angular displacement in radians from the positive \(x\)-axis, \(\text{RHO}\) is the distance from the origin to a point in the \(x\)-\(y\) plane, and \(Z\) is the height above the \(x\)-\(y\) plane. Arrays \(X\), \(Y\), and \(Z\) must be the same size (or any can be scalar).

\([\text{THETA}, \text{RHO}] = \text{cart2pol}(X, Y)\) transforms two-dimensional Cartesian coordinates stored in corresponding elements of arrays \(X\) and \(Y\) into polar coordinates.

**Algorithm**
The mapping from two-dimensional Cartesian coordinates to polar coordinates, and from three-dimensional Cartesian coordinates to cylindrical coordinates is

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

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

**See Also**
cart2sph, pol2cart, sph2cart
Purpose
Transform Cartesian coordinates to spherical

Syntax
[THETA, PHI, R] = cart2sph(X, Y, Z)

Description
[THETA, PHI, R] = cart2sph(X, Y, Z) transforms Cartesian coordinates stored in corresponding elements of arrays X, Y, and Z into spherical coordinates. Azimuth THETA and elevation PHI are angular displacements in radians measured from the positive x-axis, and the x-y plane, respectively; and R is the distance from the origin to a point.

Arrays X, Y, and Z must be the same size.

Algorithm
The mapping from three-dimensional Cartesian coordinates to spherical coordinates is

\[
\theta = \text{atan2}(y, x) \\
\phi = \text{atan2}(z, \sqrt{x^2 + y^2}) \\
r = \sqrt{x^2 + y^2 + z^2}
\]

See Also
cart2pol, pol2cart, sph2cart
**Purpose**
Case switch

**Description**
`case` is part of the `switch` statement syntax, which allows for conditional execution.

A particular case consists of the `case` statement itself followed by a case expression and one or more statements.

A case is executed only if its associated case expression (`case_expr`) is the first to match the switch expression (`switch_expr`).

**Examples**
The general form of the `switch` statement is

```plaintext
switch switch_expr
    case case_expr
        statement,...,statement
    case {case_expr1,case_expr2,case_expr3,...}
        statement,...,statement
    ...
    otherwise
        statement,...,statement
end
```

**See Also**
`switch`
**Purpose**  
Cast a variable to a different data type or class.

**Syntax**  
B = cast(A, newclass)

**Description**  
B = cast(A, newclass) casts A to class newclass. A must be convertible to class newclass. newclass must be the name of one of the built in data types.

**Example**  
a = int8(5);
b = cast(a, 'uint8');
class(b)

ans =
uint8

**See Also**  
class
Purpose

Concatenate arrays

Syntax

\[ C = \text{cat}(\text{dim}, A, B) \]
\[ C = \text{cat}(\text{dim}, A1, A2, A3, A4...) \]

Description

\( C = \text{cat}(\text{dim}, A, B) \) concatenates the arrays \( A \) and \( B \) along \( \text{dim} \).

\( C = \text{cat}(\text{dim}, A1, A2, A3, A4,...) \) concatenates all the input arrays (\( A1, A2, A3, A4, \) and so on) along \( \text{dim} \).

\( \text{cat}(2, A, B) \) is the same as \([A, B]\), and \( \text{cat}(1, A, B) \) is the same as \([A; B]\).

Remarks

When used with comma-separated list syntax, \( \text{cat}(\text{dim}, C{:}) \) or \( \text{cat}(\text{dim}, C.\text{field}) \) is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.

Examples

Given

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

concatenating along different dimensions produces

The commands

\[
A = \text{magic}(3); \ B = \text{pascal}(3);
C = \text{cat}(4, A, B);
\]

produce a 3-by-3-by-1-by-2 array.

See Also

num2cell

The special character [ ]
**Purpose**
Begin catch block

**Description**
The general form of a try statement is

```plaintext
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**
`try`, `end`, `lasterr`, `eval`, `evalin`
**caxis**

**Purpose**
Color axis scaling

**Syntax**
- `caxis([cmin cmax])`
- `caxis auto`
- `caxis manual`
- `caxis(caxis)`
- `v = caxis`
- `caxis(axes_handle,...)`

**Description**
`caxis` controls the mapping of data values to the colormap. It affects any surfaces, patches, and images with indexed CData and CDataMapping set to `scaled`. It does not affect surfaces, patches, or images with true color CData or with CDataMapping set to `direct`.

`caxis([cmin cmax])` sets the color limits to specified minimum and maximum values. Data values less than `cmin` or greater than `cmax` map to `cmin` and `cmax`, respectively. Values between `cmin` and `cmax` linearly map to the current colormap.

`caxis auto` lets MATLAB compute the color limits automatically using the minimum and maximum data values. This is the default behavior. Color values set to `Inf` map to the maximum color, and values set to `−Inf` map to the minimum color. Faces or edges with color values set to `NaN` are not drawn.

`caxis manual` and `caxis(caxis)` freeze the color axis scaling at the current limits. This enables subsequent plots to use the same limits when `hold` is `on`.

`v = caxis` returns a two-element row vector containing the `[cmin cmax]` currently in use.

`caxis(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

**Remarks**
caxis changes the `CLim` and `CLimMode` properties of axes graphics objects.

**How Color Axis Scaling Works**
Surface, patch, and image graphics objects having indexed CData and CDataMapping set to `scaled` map CData values to colors in the figure colormap each time they render. CData values equal to or less than `cmin` map to the first
color value in the colormap, and CData values equal to or greater than cmax map to the last color value in the colormap. MATLAB performs the following linear transformation on the intermediate values (referred to as C below) to map them to an entry in the colormap (whose length is m, and whose row index is referred to as index below).

\[ \text{index} = \text{fix}((C - \text{cmin})/(\text{cmax} - \text{cmin}) \times m) + 1 \]

**Examples**

Create \((X,Y,Z)\) data for a sphere and view the data as a surface.

\[
[X,Y,Z] = \text{sphere;}
C = Z;
surf(X,Y,Z,C)
\]

Values of \(C\) have the range \([-1, 1]\). Values of \(C\) near \(-1\) are assigned the lowest values in the colormap; values of \(C\) near 1 are assigned the highest values in the colormap.

To map the top half of the surface to the highest value in the color table, use

\[
caxis([-1 0])
\]

To use only the bottom half of the color table, enter

\[
caxis([-1 3])
\]

which maps the lowest CData values to the bottom of the colormap, and the highest values to the middle of the colormap (by specifying a cmax whose value is equal to cmin plus twice the range of the CData).

The command

\[
caxis auto
\]

resets axis scaling back to autoranging and you see all the colors in the surface. In this case, entering

\[
caxis
\]

returns

\[
[1 1]
\]
Adjusting the color axis can be useful when using images with scaled color data. For example, load the image data and colormap for Cape Cod, Massachusetts.

```matlab
load cape
```
This command loads the image’s data `X` and the image’s colormap `map` into the workspace. Now display the image with `CDataMapping` set to `scaled` and install the image’s colormap.

```matlab
image(X,'CDataMapping','scaled')
colormap(map)
```
MATLAB sets the color limits to span the range of the image data, which is 1 to 192:

```matlab
caxis
ans =
   1   192
```
The blue color of the ocean is the first color in the colormap and is mapped to the lowest data value (1). You can effectively move sea level by changing the lower color limit value. For example,
See Also

axes, axis, colormap, get, mesh, pcolor, set, surf
The CLim and CLimMode properties of axes graphics objects
The Colormap property of figure graphics objects
“Color Operations” for related functions
Axes Color Limits for more examples
Purpose
Change working directory

Graphical Interface
As an alternative to the cd function, use the current directory field in the MATLAB desktop toolbar.

Syntax

```
cd
w = cd
cd('directory')
cd('..')
cd directory or cd ..
```

Description

`cd` displays the current working directory.

```
w = cd assigns the current working directory to w.
```

`cd('directory')` sets the current working directory to `directory`. Use the full pathname for `directory`. On UNIX platforms, the character ~ is interpreted as the user’s root directory.

`cd('..')` changes the current working directory to the directory above it.

`cd directory or cd ..` is the unquoted form of the syntax.

Examples

On UNIX

```
cd('/usr/local/matlab/toolbox/demos')
```
changes the current working directory to demos.

On Windows

```
cd('c:/toolbox/matlab/demos')
```
changes the current working directory to demos. Then typing

```
cd ..
```
changes the current working directory to matlab.

See Also
dir, fileparts, mfilename, path, pwd, what
cd (ftp)

**Purpose**
Change current directory on FTP server

**Syntax**

```
cd(f)
cd(f,'dirname')
cd(f,'..')
```

**Description**

`cd(f)` Displays the current directory on the FTP server `f`, where `f` was created using `ftp`.

`cd(f,'dirname')` Changes the current directory on the FTP server `f` to `dirname`, where `f` was created using `ftp`. After running `cd`, the object `f` remembers the current directory on the FTP server. You can then perform file operations functions relative to `f` using the methods `delete`, `dir`, `mget`, `mkdir`, `mput`, `rename`, and `rmdir`.

`cd(f,'..')` changes the current directory on the FTP server `f` to the directory above the current one.

**Examples**

Connect to the MathWorks FTP server.

```
tmw=ftp('ftp.mathworks.com');
```

View the contents.

```
dir(tmw)
```

```
.                        incoming      pickup
 README                   matlab        pub
 README.incoming         outgoing     pubs
```

Change the current directory to `pub`.

```
cd(tmw,'pub');
```
View the contents of pub.

```
dir(tmw)
```

```
.       bin       digest       matweb.exe   proceedings
..      books      doc       ops       product-info
INDEX   compiler   france     outgoing    tech-support
NEWFILES conference ftphelp   patch       temp
admin    connections ls-lR   pentium     utilities
beta     contrib   mathworks pressroom
```

**See Also**  
dir (ftp), ftp
**Purpose**
Convert complex diagonal form to real block diagonal form

**Syntax**

\[ [V,D] = \text{cdf2rdf}(V,D) \]

**Description**
If the eigensystem \( [V,D] = \text{eig}(X) \) has complex eigenvalues appearing in complex-conjugate pairs, \text{cdf2rdf} transforms the system so \( D \) is in real diagonal form, with 2-by-2 real blocks along the diagonal replacing the complex pairs originally there. The eigenvectors are transformed so that

\[ X = V*D/V \]

continues to hold. The individual columns of \( V \) are no longer eigenvectors, but each pair of vectors associated with a 2-by-2 block in \( D \) spans the corresponding invariant vectors.

**Examples**
The matrix

\[
X = \\
\begin{bmatrix}
1 & 2 & 3 \\
0 & 4 & 5 \\
0 & -5 & 4 \\
\end{bmatrix}
\]

has a pair of complex eigenvalues.

\[ [V,D] = \text{eig}(X) \]

\[
V = \\
\begin{bmatrix}
1.0000 & -0.0191 - 0.4002i & -0.0191 + 0.4002i \\
0 & 0 - 0.6479i & 0 + 0.6479i \\
0 & 0.6479 & 0.6479 \\
\end{bmatrix}
\]

\[
D = \\
\begin{bmatrix}
1.0000 & 0 & 0 \\
0 & 4.0000 + 5.0000i & 0 \\
0 & 0 & 4.0000 - 5.0000i \\
\end{bmatrix}
\]

Converting this to real block diagonal form produces

\[ [V,D] = \text{cdf2rdf}(V,D) \]
The real diagonal form for the eigenvalues is obtained from the complex form using a specially constructed similarity transformation.

See Also eig, rsf2csf
Purpose
Construct a cdfepoch object for Common Data Format (CDF) export

Syntax
E = cdfepoch(date)

Description
E = cdfepoch(date) constructs a cdfepoch object, where date is a valid string (datestr), a number (datenum) representing a date, or a cdfepoch object.

When writing data to a CDF using cdfwrite, use cdfepoch to convert MATLAB formatted dates to CDF formatted dates. The MATLAB cdfepoch object simulates the CDFEPOCH data type in CDF files.

Note
A CDF epoch is the number of milliseconds since 1-Jan-0000. MATLAB datenums are the number of days since 0-Jan-0000.

See Also
cdfinfo, cdfread, cdfwrite, datenum
**Purpose**
Return information about a CDF file

**Syntax**
info = cdfinfo(file)

**Description**
info = cdfinfo(file) returns information about the Common Data Format (CDF) file specified in the string file.

**Note** Because cdfinfo creates temporary files, the current working directory must be writeable.

The return value, info, is a structure that contains the fields listed alphabetically in the following table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileModDate</td>
<td>Text string indicating the date the file was last modified</td>
</tr>
<tr>
<td>Filename</td>
<td>Text string specifying the name of the file</td>
</tr>
<tr>
<td>FileSettings</td>
<td>Structure array containing library settings used to create the file</td>
</tr>
<tr>
<td>FileSize</td>
<td>Double scalar specifying the size of the file, in bytes</td>
</tr>
<tr>
<td>Format</td>
<td>Text string specifying the file format</td>
</tr>
<tr>
<td>FormatVersion</td>
<td>Text string specifying the version of the CDF library used to create the file</td>
</tr>
<tr>
<td>GlobalAttributes</td>
<td>Structure array that contains one field for each global attribute. The name of each field corresponds to the name of an attribute. The data in each field, contained in a cell array, represents the entry values for that attribute.</td>
</tr>
<tr>
<td>Subfiles</td>
<td>Filenames containing the CDF file’s data, if it is a multifile CDF</td>
</tr>
</tbody>
</table>
### VariableAttributes

Structure array that contains one field for each variable attribute. The name of each field corresponds to the name of an attribute. The data in each field is contained in a \( n \)-by-2 cell array, where \( n \) is the number of variables. The first column of this cell array contains the variable names associated with the entries. The second column contains the entry values.

### Variables

N-by-6 cell array, where \( N \) is the number of variables, containing information about the variables in the file. The columns present the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Text string specifying name of variable</td>
</tr>
<tr>
<td>2</td>
<td>Double array specifying the dimensions of the variable, as returned by the \texttt{size} function</td>
</tr>
<tr>
<td>3</td>
<td>Double scalar specifying the number of records assigned for the variable</td>
</tr>
<tr>
<td>4</td>
<td>Text string specifying the data type of the variable, as stored in the CDF file</td>
</tr>
<tr>
<td>5</td>
<td>Text string specifying the record and dimension variance settings for the variable. The single ( T ) or ( F ) to the left of the slash designates whether values vary by record. The zero or more ( T ) or ( F ) letters to the right of the slash designate whether values vary at each dimension. Here are some examples.</td>
</tr>
<tr>
<td>6</td>
<td>Text string specifying the sparsity of the variable’s records, with these possible values:</td>
</tr>
</tbody>
</table>

| T/     | (scalar variable) |
| F/T    | (one-dimensional variable) |
| T/TFF  | (three-dimensional variable) |

| 'Full'  |
| 'Sparse (padded)' |
| 'Sparse (nearest)' |
Note  Attribute names returned by *cdfinfo* might not match the names of the attributes in the CDF file exactly. Attribute names can contain characters that are illegal in MATLAB field names. *cdfinfo* removes illegal characters that appear at the beginning of attributes and replaces other illegal characters with underscores ('_'). When *cdfinfo* modifies an attribute name, it appends the attribute's internal number to the end of the field name. For example, the attribute name `Variable%Attribute` becomes `Variable_Attribute_013`.

Examples

```matlab
info = cdfinfo('example.cdf')
info =
    Filename: 'example.cdf'
    FileModDate: '29-Jun-1995 05:51:58'
    FileSize: 230513
    Format: 'CDF'
    FormatVersion: '2.4.8'
    FileSettings: [1x1 struct]
    Subfiles: {}
    Variables: {7x6 cell}
    GlobalAttributes: [1x1 struct]
    VariableAttributes: [1x1 struct]
```

```matlab
info.Variables
ans =
    'L_gse' [1x2 double] [  1] 'char' 'F/T' 'Full'
    'Status%C1' [1x2 double] [7493] 'uint8' 'T/T' 'Full'
    'B_gse%C1' [1x2 double] [7493] 'single' 'T/' 'Full'
    'B_nsigma%C1' [1x2 double] [7493] 'single' 'T/' 'Full'
```

See Also

cdfread
Purpose
Read data from a CDF file

Syntax
data = cdfread(file)
data = cdfread(file, 'records', recnums, ...)
data = cdfread(file, 'variables', varnames, ...)
data = cdfread(file, 'slices', dimensionvalues, ...)
[data, info] = cdfread(file, ...)

Description
data = cdfread(file) reads all the variables from each record of the Common Data Format (CDF) file specified in the string file. The return value data is a cell array in which each row contains a record and each column represents a variable. See the Examples section for an illustration.

Note Because cdfread creates temporary files, the current working directory must be writeable.

data = cdfread(file, 'records', recnums, ...) reads only those records specified in the vector recnums. The record numbers are zero based. The return value data is a cell array having length(recnums) number of rows and as many columns as there are variables.

data = cdfread(file, 'variables', varnames, ...) reads only those variables specified in the 1-by-N or N-by-1 cell array of strings varnames. The return value data is returned in a cell array having length(varnames) number of columns and a row for each record requested.

data = cdfread(file, 'slices', dimensionvalues, ...) reads specific values from the records of one variable in the CDF file. The N-by-3 matrix dimensionvalues indicates which records are to be read by specifying start, interval, and count parameters for each of the N dimensions of the variable. The start parameter is zero based.

The number of rows in dimensionvalues must be less than or equal to the number of dimensions of the variable. Unspecified rows default to [0 1 N], where N is the total number of values in a record. This causes cdfread to read every value from those dimensions.
Because you can read just one variable at a time, you must also include a 'variables' parameter with this syntax.

```
[data, info] = cdfread(file, ...) also returns details about the CDF file in the info structure.
```

### Examples

Read all the data from the file.
```
data = cdfread('example.cdf');
```

Read just the data from variable 'Time'.
```
data = cdfread('example.cdf', 'Variable', {'Time'});
```

Read the first value in the first dimension, the second value in the second dimension, the first and third values in the third dimension, and all values in the remaining dimension of the variable 'multidimensional'.
```
data = cdfread('example.cdf', 'Variable', ...
    {'multidimensional'}, 'Slices', [0 1 1; 1 1 1; 0 2 2]);
```

This is similar to reading the whole variable into 'data' and then using the MATLAB command
```
data{1}(1, 2, [1 3], :)
```

### See Also

cdfinfo, cdfwrite, cdfepoch
Purpose
Write data to a CDF file

Syntax
```matlab
cdfwrite(file, variablelist)
cdfwrite(..., 'PadValues', padvals)
cdfwrite(..., 'GlobalAttributes', gattrib)
cdfwrite(..., 'VariableAttributes', vattrib)
cdfwrite(..., 'WriteMode', mode)
cdfwrite(..., 'Format', format)
```

Description
cdfwrite(file, variablelist) writes out a Common Data Format (CDF) file, specified in the string file. The variablelist argument is a cell array of ordered pairs, each of which comprises a CDF variable name (a string) and the corresponding CDF variable value. To write out multiple records for a variable, put the values in a cell array where each element in the cell array represents a record.

Note Because cdfwrite creates temporary files, both the destination directory for the file and the current working directory must be writeable.

cdfwrite(..., 'PadValues', padvals) writes out pad values for given variable names. padvals is a cell array of ordered pairs, each of which comprises a variable name (a string) and a corresponding pad value. Pad values are the default values associated with the variable when an out-of-bounds record is accessed. Variable names that appear in padvals must appear in variablelist.

cdfwrite(..., 'GlobalAttributes', gattrib) writes the structure gattrib as global metadata for the CDF file. Each field of the structure is the name of a global attribute. The value of each field contains the value of the attribute. To write out multiple values for an attribute, put the values in a cell array where each element in the cell array represents a record.

Note To specify a global attribute name that is illegal in MATLAB, create a field called 'CDFAttributeRename' in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered
pair consists of the name of the original attribute, as listed in the GlobalAttributes structure, and the corresponding name of the attribute to be written to the CDF file.

cdfwrite(..., 'VariableAttributes', vattrib) writes the structure vattrib as variable metadata for the CDF. Each field of the struct is the name of a variable attribute. The value of each field should be an M-by-2 cell array where M is the number of variables with attributes. The first element in the cell array should be the name of the variable and the second element should be the value of the attribute for that variable.

**Note** To specify a variable attribute name that is illegal in MATLAB, create a field called 'CDFAttributeRename' in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the VariableAttributes struct, and the corresponding name of the attribute to be written to the CDF file. If you are specifying a variable attribute of a CDF variable that you are renaming, the name of the variable in the VariableAttributes structure must be the same as the renamed variable.

cdfwrite(...,'WriteMode',mode), where mode is either 'overwrite' or 'append', indicates whether or not the specified variables should be appended to the CDF file if the file already exists. By default, cdfwrite overwrites existing variables and attributes.

cdfwrite(...,'Format',format), where format is either 'multifile' or 'singlefile', indicates whether or not the data is written out as a multifile CDF. In a multifile CDF, each variable is stored in a separate file with the name *.vN, where N is the number of the variable that is written out to the CDF. By default, cdfwrite writes out a single file CDF. When 'WriteMode' is set to 'Append', the 'Format' option is ignored, and the format of the preexisting CDF is used.

**Examples**

Write out a file 'example.cdf' containing a variable 'Longitude' with the value [0:360].
cdfwrite('example', {'Longitude', 0:360});

Write out a file 'example.cdf' containing variables 'Longitude' and 'Latitude' with the variable 'Latitude' having a pad value of 10 for all out-of-bounds records that are accessed.

cdfwrite('example', {'Longitude', 0:360, 'Latitude', 10:20},... 'PadValues', {'Latitude', 10});

Write out a file 'example.cdf', containing a variable 'Longitude' with the value [0:360], and with a variable attribute of 'validmin' with the value 10.

varAttribStruct.validmin = {'longitude' [10]};
cdfwrite('example', {'Longitude' 0:360}, 'VarAttribStruct',... varAttribStruct);

See Also

cdfread, cdfinfo, cdfepoch
Purpose
Round toward infinity

Syntax
B = ceil(A)

Description
B = ceil(A) rounds the elements of A to the nearest integers greater than or equal to A. For complex A, the imaginary and real parts are rounded independently.

Examples
a = [-1.9, -0.2, 3.4, 5.6, 7, 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

ceil(a)

ans =
Columns 1 through 4
-1.0000    0    4.0000    6.0000

Columns 5 through 6
7.0000    3.0000 + 4.0000i

See Also
fix, floor, round
Purpose
Create cell array

Syntax
\[
c = \text{cell}(n)
\]
\[
c = \text{cell}(m,n) \text{ or } c = \text{cell}([m \ n])
\]
\[
c = \text{cell}(m,n,p,...) \text{ or } c = \text{cell}([m \ n \ p ...])
\]
\[
c = \text{cell(size(A))}
\]
\[
c = \text{cell(javaobj)}
\]

Description
\[
c = \text{cell}(n) \text{ creates an } n\text{-by-}n \text{ cell array of empty matrices. An error message appears if } n \text{ is not a scalar.}
\]
\[
c = \text{cell}(m,n) \text{ or } c = \text{cell}([m,n]) \text{ creates an } m\text{-by-}n \text{ cell array of empty matrices. Arguments } m \text{ and } n \text{ must be scalars.}
\]
\[
c = \text{cell}(m,n,p,...) \text{ or } c = \text{cell}([m \ n \ p ...]) \text{ creates an } m\text{-by-}n\text{-by-}p\text{-... cell array of empty matrices. Arguments } m, n, p, ... \text{ must be scalars.}
\]
\[
c = \text{cell(size(A))} \text{ creates a cell array the same size as } A \text{ containing all empty matrices.}
\]
\[
c = \text{cell(javaobj)} \text{ converts a Java array or Java object javaobj into a MATLAB cell array. Elements of the resulting cell array will be of the MATLAB type (if any) closest to the Java array elements or Java object.}
\]

Examples
This example creates a cell array that is the same size as another array, A.
\[
A = \text{ones(2,2)}
\]
\[
A =
\begin{bmatrix}
1 & 1 \\
1 & 1
\end{bmatrix}
\]
\[
c = \text{cell(size(A))}
\]
\[
c =
\begin{bmatrix}
[] & [] \\
[] & []
\end{bmatrix}
\]
The next example converts an array of java.lang.String objects into a MATLAB cell array.
strArray = java_array('java.lang.String',3);
strArray(1) = java.lang.String('one');
strArray(2) = java.lang.String('two');
strArray(3) = java.lang.String('three');

cellArray = cell(strArray)
cellArray =
    'one'
    'two'
    'three'

See Also

num2cell, ones, rand, randn, zeros
Purpose
Convert cell array of matrices into single matrix

Syntax
\[ m = \text{cell2mat}(c) \]

Description
\( m = \text{cell2mat}(c) \) converts a multidimensional cell array \( c \) with contents of the same data type into a single matrix, \( m \). The contents of \( c \) must be able to concatenate into a hyperrectangle. Moreover, for each pair of neighboring cells, the dimensions of the cells’ contents must match, excluding the dimension in which the cells are neighbors.

The example shown below combines matrices in a 3-by-2 cell array into a single 60-by-50 matrix:

\[
\text{cell2mat}(c)
\]

Remarks
The dimensionality (or number of dimensions) of \( m \) will match the highest dimensionality contained in the cell array.

\text{cell2mat} is not supported for cell arrays containing cell arrays or objects.

Examples
Combine the matrices in four cells of cell array \( C \) into the single matrix, \( M \):

\[
C = \{[1 \ 2 \ 3 \ 4]; \ [5; \ 9] \ [6 \ 7 \ 8; \ 10 \ 11 \ 12]\}
\]

\[
C =
\begin{bmatrix}
[1] & [1x3 \ double] \\
[2x1 \ double] & [2x3 \ double]
\end{bmatrix}
\]
cell2mat

C{1,1}                     C{1,2}
ans =                      ans =
  1                          2     3     4
C{2,1}                     C{2,2}
ans =                      ans =
  5                          6     7     8
   9                         10    11    12

M = cell2mat(C)
M =
  1     2     3     4
  5     6     7     8
  9    10    11    12

See Also  mat2cell, num2cell
Purpose

Convert cell array to structure array

Syntax

s = cell2struct(c, fields, dim)

Description

s = cell2struct(c, fields, dim) creates a structure array s from the information contained within cell array c.

The fields argument specifies field names for the structure array. fields can be a character array or a cell array of strings.

The dim argument controls which axis of the cell array is to be used in creating the structure array. The length of c along the specified dimension must match the number of fields named in fields. In other words, the following must be true.

\[
\text{size}(c,\text{dim}) = \text{length(fields)} \quad \% \text{ if fields is a cell array}
\]

\[
\text{size}(c,\text{dim}) = \text{size(fields,1)} \quad \% \text{ if fields is a char array}
\]

Examples

The cell array c in this example contains information on trees. The three columns of the array indicate the common name, genus, and average height of a tree.

\[
c = \{'\text{birch}', '\text{betula}', 65; '\text{maple}', '\text{acer}', 50\}
\]

\[
c =
\begin{array}{ccc}
\text{birch} & \text{betula} & 65 \\
\text{maple} & \text{acer} & 50
\end{array}
\]

To put this information into a structure with the fields name, genus, and height, use cell2struct along the second dimension of the 2-by-3 cell array.

\[
\text{fields} = \{'\text{name}', '\text{genus}', '\text{height}'\};
\]

\[
s = \text{cell2struct}(c, \text{fields}, 2);
\]

This yields the following 2-by-1 structure array.

\[
s(1)
\begin{array}{lll}
\text{name}: \text{birch} & \text{genus}: \text{betula} & \text{height}: 65
\end{array}
\]

\[
s(2)
\begin{array}{lll}
\text{name}: \text{maple} & \text{genus}: \text{acer} & \text{height}: 50
\end{array}
\]
See Also

struct2cell, cell, iscell, struct, isstruct, fieldnames, dynamic field names
Purpose
Display cell array contents.

Syntax
celldisp(C)

celldisp(C, name)

Description
celldisp(C) recursively displays the contents of a cell array.

celldisp(C, name) uses the string name for the display instead of the name of
the first input (or ans).

Example
Use celldisp to display the contents of a 2-by-3 cell array:

C = {{[1 2] 'Tony' 3+4i; [1 2;3 4] -5 'abc'}};
celldisp(C)

C{1,1} =
 1 2

C{2,1} =
 1 2
 3 4

C{1,2} =
Tony

C{2,2} =
-5

C{1,3} =
3.0000+ 4.0000i

C{2,3} =
abc

See Also
cellplot
cellfun

**Purpose**  
Apply a function to each element in a cell array

**Syntax**  
D = cellfun('fname',C)  
D = cellfun('size',C,k)  
D = cellfun('isclass',C,classname)

**Description**  
D = cellfun('fname',C) applies the function fname to the elements of the cell array C and returns the results in the double array D. Each element of D contains the value returned by fname for the corresponding element in C. The output array D is the same size as the cell array C.

These functions are supported:

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>isempty</td>
<td>true for an empty cell element</td>
</tr>
<tr>
<td>islogical</td>
<td>true for a logical cell element</td>
</tr>
<tr>
<td>isreal</td>
<td>true for a real cell element</td>
</tr>
<tr>
<td>length</td>
<td>Length of the cell element</td>
</tr>
<tr>
<td>ndims</td>
<td>Number of dimensions of the cell element</td>
</tr>
<tr>
<td>prodofsize</td>
<td>Number of elements in the cell element</td>
</tr>
</tbody>
</table>

D = cellfun('size',C,k) returns the size along the kth dimension of each element of C.

D = cellfun('isclass',C,'classname') returns true for each element of C that matches classname. This function syntax returns false for objects that are a subclass of classname.

**Limitations**  
If the cell array contains objects, cellfun does not call overloaded versions of the function fname.

**Example**  
Consider this 2-by-3 cell array:

\[
C\{1,1\} = [1 \ 2; \ 4 \ 5];  
C\{1,2\} = 'Name';
\]
C{1,3} = pi;
C{2,1} = 2 + 4i;
C{2,2} = 7;
C{2,3} = magic(3);

cellfun returns a 2-by-3 double array:

D = cellfun('isreal',C)

D =
    1  1  1
    0  1  1

len = cellfun('length',C)

len =
    2  4  1
    1  1  3

isdbl = cellfun('isclass',C,'double')

isdbl =
    1  0  1
    1  1  1

See Also
iseempty, islogical, isreal, length, ndims, size
**Purpose**
Graphically display the structure of cell arrays

**Syntax**
```
cellplot(c)
cellplot(c,'legend')
handles = cellplot(...)
```

**Description**
`cellplot(c)` displays a figure window that graphically represents the contents of `c`. Filled rectangles represent elements of vectors and arrays, while scalars and short text strings are displayed as text.

`cellplot(c, 'legend')` also puts a legend next to the plot.

`handles = cellplot(c)` displays a figure window and returns a vector of surface handles.

**Limitations**
The `cellplot` function can display only two-dimensional cell arrays.

**Examples**
Consider a 2-by-2 cell array containing a matrix, a vector, and two text strings:
```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces
cellstr

Purpose
Create cell array of strings from character array

Syntax
\[ c = \text{cellstr}(S) \]

Description
\( c = \text{cellstr}(S) \) places each row of the character array \( S \) into separate cells of \( c \). Use the char function to convert back to a string matrix.

Examples
Given the string matrix
\[
S = ['abc ';'defg';'hi ']
\]

\[
S = \begin{array}{c}
abc \\
defg \\
hi \\
\end{array}
\]

whos S

\[
\begin{array}{cccc}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} \\
S & 3x4 & 24 & \text{char array}
\end{array}
\]

The following command returns a 3-by-1 cell array.

\[
c = \text{cellstr}(S)
\]

\[
c = \begin{array}{c}
'abc' \\
'defg' \\
'hi'
\end{array}
\]

whos c

\[
\begin{array}{cccc}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} \\
c & 3x1 & 294 & \text{cell array}
\end{array}
\]

See Also
iscellstr, strings
Purpose
Conjugate Gradients Squared method

Syntax

\[
x = \text{cgs}(A,b) \\
cgs(A,b,tol) \\
cgs(A,b,tol,maxit) \\
cgs(A,b,tol,maxit,M) \\
cgs(A,b,tol,maxit,M1,M2) \\
cgs(A,b,tol,maxit,M1,M2,x0) \\
cgs(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...) \\
[x,flag] = \text{cgs}(A,b,...) \\
[x,flag,relres] = \text{cgs}(A,b,...) \\
[x,flag,relres,iter] = \text{cgs}(A,b,...) \\
[x,flag,relres,iter,resvec] = \text{cgs}(A,b,...)
\]

Description

\[x = \text{cgs}(A,b)\] attempts to solve the system of linear equations \(A\times x = b\) for \(x\). The \(n\)-by-\(n\) coefficient matrix \(A\) must be square and should be large and sparse. The column vector \(b\) must have length \(n\). \(A\) can be a function \(afun\) such that \(afun(x)\) returns \(A\times x\).

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

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

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

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

\(\text{cgs}(A,b,tol,maxit,M1,M2,x0)\) specifies the initial guess \(x0\). If \(x0\) is [], then \(\text{cgs}\) uses the default, an all-zero vector.
cgs(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] = cgs(A,b,...) returns a solution x and a flag that describes the convergence of cgs.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cgs converged to the desired tolerance tol within maxit iterations.</td>
</tr>
<tr>
<td>1</td>
<td>cgs 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>cgs stagnated. (Two consecutive iterates were the same.)</td>
</tr>
<tr>
<td>4</td>
<td>One of the scalar quantities calculated during cgs 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] = cgs(A,b,...) also returns the relative residual \( \text{norm}(b-A\times x)/\text{norm}(b) \). If flag is 0, then relres \( \leq \) tol.

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

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

**Examples**

**Example 1.**

```matlab
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;  maxit = 15;
M1 = diag([10:-1:1 1:10]);
x = cgs(A,b,tol,maxit,M1,[],[]);
```
Alternatively, use this matrix-vector product function

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

and this preconditioner backsolve function

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

as inputs to `cgs`.

```matlab
x1 = cgs(@afun,b,tol,maxit,@mfun,[],[],21);
```

Note that both `afun` and `mfun` must accept `cgs`'s extra input `n=21`.

**Example 2.**

```matlab
load west0479
A = west0479
b = sum(A,2)
[x,flag] = cgs(A,b)
```

`flag` is 1 because `cgs` does not converge to the default tolerance $1e^{-6}$ within the default 20 iterations.

```matlab
[L1,U1] = luinc(A,1e-5)
[x1,flag1] = cgs(A,b,1e-6,20,L1,U1)
```

`flag1` is 2 because the upper triangular $U_1$ has a zero on its diagonal, and `cgs` fails in the first iteration when it tries to solve a system such as $U_1y = r$ for $y$ with backslash.

```matlab
[L2,U2] = luinc(A,1e-6)
[x2,flag2,relres2,iter2,resvec2] = cgs(A,b,1e-15,10,L2,U2)
```

`flag2` is 0 because `cgs` converges to the tolerance of $6.344e^{-16}$ (the value of `relres2`) at the fifth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of $1e^{-6}$.

`resvec2(1) = norm(b)` and `resvec2(6) = norm(b-A*x2)`. You can follow the
progress of cg

gs by plotting the relative residuals at each iteration starting from
the initial estimate (iterate number 0) with

```matlab
semilogx(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```

![Graph showing relative residual vs iteration number](image)

**See Also**
bicg, bicgstab, gmres, lsqr, luinc, minres, pcg, qmr, symmlq
@ (function handle), \ (backslash)

**References**

**char**

**Purpose**
Create character array (string)

**Syntax**

\[
S = \text{char}(X) \\
S = \text{char}(C) \\
S = \text{char}(t1,t2,t3...) \\
\]

**Description**

\(S = \text{char}(X)\) converts the array \(X\) that contains positive integers representing character codes into a MATLAB character array (the first 127 codes are ASCII). The actual characters displayed depend on the character set encoding for a given font. The result for any elements of \(X\) outside the range from 0 to 65535 is not defined (and can vary from platform to platform). Use `double` to convert a character array into its numeric codes.

\(S = \text{char}(C)\), when \(C\) is a cell array of strings, places each element of \(C\) into the rows of the character array \(s\). Use `cellstr` to convert back.

\(S = \text{char}(t1,t2,t3,...)\) forms the character array \(S\) containing the text strings \(T1,T2,T3,...\) as rows, automatically padding each string with blanks to form a valid matrix. Each text parameter, \(T_i\), can itself be a character array. This allows the creation of arbitrarily large character arrays. Empty strings are significant.

**Remarks**

Ordinarily, the elements of \(A\) are integers in the range 32:127, which are the printable ASCII characters, or in the range 0:255, which are all 8-bit values. For noninteger values, or values outside the range 0:255, the characters printed are determined by `fix(rem(A,256))`.

**Examples**

To print a 3-by-32 display of the printable ASCII characters,

\[
\text{ascii} = \text{char}(\text{reshape}(32:127,32,3))' \\
\text{ascii} = \\
! # $ % & ' ( ) *+ , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? \@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ' a b c d e f g h i j k l m n o p q r s t u v w x y z { | } - \\
\]

2-334
See Also

cellstr, double, get, set, strings, strvcat, text
checkin

**Purpose**
Check file into source control system

**Graphical Interface**
As an alternative to the `checkin` function, use **Source Control Check In** in the Editor, Simulink, or Stateflow **File** menu.

**Syntax**
```
checkin('filename','comments','string')
checkin({{filename1','filename2','filename3', ...},'comments', 'string'})
checkin('filename','option','value', ...)
```

**Description**
`checkin('filename','comments','string')` checks in the file named `filename` to the source control system. Use the full pathname for the `filename`. You must save the file before checking it in. The file can be open or closed when you use `checkin`. The string argument is a MATLAB string containing check-in comments for the source control system. You must supply the `comments` argument and `'string'`.

`checkin({{filename1','filename2','filename3', ...},'comments', 'string'})` checks in the files named `filename1` through `filenamen` to the source control system. Use the full pathnames for the files. Additional arguments apply to all files checked in.

`checkin('filename','option','value', ...)` provides additional `checkin` options. The `option` and `value` arguments are shown in the table below.

<table>
<thead>
<tr>
<th>option Argument</th>
<th>Purpose</th>
<th>value Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>'force'</td>
<td>When set to on, <code>filename</code> is checked in even if the file has not changed since it was checked out. The default value for <code>force</code> is <code>off</code>.</td>
<td>'on' 'off' (default)</td>
</tr>
<tr>
<td>'lock'</td>
<td>When set to on, <code>filename</code> remains checked out. Comments are submitted. The default value for <code>lock</code> is <code>off</code>.</td>
<td>'on' 'off' (default)</td>
</tr>
</tbody>
</table>

You can check in a file that you checked out in a previous MATLAB session or that you checked out directly from your source control system.
Examples

Check in a File with Comments

Typing

    checkin('/matlab/mymfiles/clock.m','comments','Adjustment for Y2K')

checks in the file /matlab/mymfiles/clock.m to the source control system with the comment Adjustment for Y2K.

Check in Multiple Files with Comments

Typing

    checkin({'/matlab/mymfiles/clock.m', ...
    '/matlab/mymfiles/calendar.m'},'comments','Adjustment for Y2K')

checks two files into the source control system using the same comment for each.

Check a File in and Keep It Checked out

Typing

    checkin('/matlab/mymfiles/clock.m','comments','Adjustment for Y2K','lock','on')

checks the file /matlab/mymfiles/clock.m into the source control system and keeps the file checked out.

See Also

checkout, cmopts, undocheckout
check out

Purpose
Check file out of source control system

Graphical Interface
As an alternative to the checkout function, use Source Control Check Out in the Editor, Simulink, or Stateflow File menu.

Syntax
checkout('filename')
checkout({'filename1','filename2','filename3', ...})
checkout('filename','option','value', ...)

Description
checkout('filename') checks out the file named filename from the source control system. filename must be the full pathname for the file. The file can be open or closed when you use checkout.

checkout({'filename1','filename2','filename3', ...}) checks out the files named filename1 through filenamen from the source control system. Use the full pathnames for the files. Additional arguments apply to all files checked out.

checkout('filename','option','value', ...) provides additional checkout options. The option and value arguments are shown in the following table.
If you end the MATLAB session, the file remains checked out. You can check in the file from within MATLAB during a later session, or directly from your source control system.

### Examples

#### Check out a File

Typing

```matlab
checkout('/matlab/mymfiles/clock.m')
```

checks out the file `/matlab/mymfiles/clock.m` from the source control system.
**checkout**

**Check out Multiple Files**
Typing
```matlab
classpath(['/matlab/mymfiles/clock.m','...'/matlab/mymfiles/calendar.m'])
```
checks out `/matlab/mymfiles/clock.m` and `/matlab/mymfiles/calendar.m` from the source control system.

**Force a Checkout, Even If File Is Already Checked out**
Typing
```matlab
classpath('/matlab/mymfiles/clock.m','force','on')
```
checks out `/matlab/mymfiles/clock.m` even if `clock.m` is already checked out to you.

**Check out Specified Revision of File**
Typing
```matlab
classpath('/matlab/mymfiles/clock.m','revision','1.1')
```
checks out revision 1.1 of `clock.m`.

**See Also**
checkin, cmopts, undocheckout
**Purpose**

Cholesky factorization

**Syntax**

\[ \text{R} = \text{chol}(\text{X}) \]

\[ [\text{R},\text{p}] = \text{chol}(\text{X}) \]

**Description**

The \text{chol} function uses only the diagonal and upper triangle of \( X \). The lower triangular is assumed to be the (complex conjugate) transpose of the upper. That is, \( X \) is Hermitian.

\( \text{R} = \text{chol}(\text{X}) \), where \( X \) is positive definite produces an upper triangular \( R \) so that \( R' \cdot R = X \). If \( X \) is not positive definite, an error message is printed.

\[ [\text{R},\text{p}] = \text{chol}(\text{X}) \], with two output arguments, never produces an error message. If \( X \) is positive definite, then \( p = 0 \) and \( R \) is the same as above. If \( X \) is not positive definite, then \( p \) is a positive integer and \( R \) is an upper triangular matrix of order \( q = p - 1 \) so that \( R' \cdot R = X(1:q,1:q) \).

**Examples**

The binomial coefficients arranged in a symmetric array create an interesting positive definite matrix.

\begin{verbatim}
 n = 5; 
 X = pascal(n) 
 X =
 1 1 1 1 1 
 1 2 3 4 5 
 1 3 6 10 15 
 1 4 10 20 35 
 1 5 15 35 70 
\end{verbatim}

It is interesting because its Cholesky factor consists of the same coefficients, arranged in an upper triangular matrix.

\begin{verbatim}
 R = chol(X) 
 R =
 1 1 1 1 1 
 0 1 2 3 4 
 0 0 1 3 6 
 0 0 0 1 4 
 0 0 0 0 1 
\end{verbatim}
Destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element.

\[ X(n,n) = X(n,n) - 1 \]

\[
X = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 \\
1 & 3 & 6 & 10 & 15 \\
1 & 4 & 10 & 20 & 35 \\
1 & 5 & 15 & 35 & 69 \\
\end{bmatrix}
\]

Now an attempt to find the Cholesky factorization fails.

**Algorithm**

**Inputs of Type Double**

For inputs of type double, chol uses the the LAPACK subroutines DPOTRF (real) and ZPOTRF (complex).

**Inputs of Type Single**

For inputs of type single, chol uses the the LAPACK subroutines SPOTRF (real) and CPOTRF (complex).

**References**


**See Also**

cholinc, cholupdate
Purpose
Sparse incomplete Cholesky and Cholesky-Infinity factorizations

Syntax
R = cholinc(X,droptol)
R = cholinc(X,options)
R = cholinc(X,'0')
[R,p] = cholinc(X,'0')
R = cholinc(X,'inf')

Description
cholinc produces two different kinds of incomplete Cholesky factorizations: the drop tolerance and the 0 level of fill-in factorizations. These factors may be useful as preconditioners for a symmetric positive definite system of linear equations being solved by an iterative method such as pcg (Preconditioned Conjugate Gradients). cholinc works only for sparse matrices.

R = cholinc(X,droptol) performs the incomplete Cholesky factorization of X, with drop tolerance droptol.

R = cholinc(X,options) allows additional options to the incomplete Cholesky factorization. options is a structure with up to three fields:

droptol Drop tolerance of the incomplete factorization
michol Modified incomplete Cholesky
rdiag Replace zeros on the diagonal of R

Only the fields of interest need to be set.

droptol is a non-negative scalar used as the drop tolerance for the incomplete Cholesky factorization. This factorization is computed by performing the incomplete LU factorization with the pivot threshold option set to 0 (which forces diagonal pivoting) and then scaling the rows of the incomplete upper triangular factor, U, by the square root of the diagonal entries in that column. Since the nonzero entries U(i,j) are bounded below by droptol*norm(X(:,j)) (see luinc), the nonzero entries R(i,j) are bounded below by the local drop tolerance droptol*norm(X(:,j))/R(i,i).

Setting droptol = 0 produces the complete Cholesky factorization, which is the default.
cholinc

michol stands for modified incomplete Cholesky factorization. Its value is either 0 (unmodified, the default) or 1 (modified). This performs the modified incomplete LU factorization of X and scales the returned upper triangular factor as described above.

rdiag is either 0 or 1. If it is 1, any zero diagonal entries of the upper triangular factor R are replaced by the square root of the local drop tolerance in an attempt to avoid a singular factor. The default is 0.

R = cholinc(X,'0') produces the incomplete Cholesky factor of a real sparse matrix that is symmetric and positive definite using no fill-in. The upper triangular R has the same sparsity pattern as \text{triu}(X), although R may be zero in some positions where X is nonzero due to cancellation. The lower triangle of X is assumed to be the transpose of the upper. Note that the positive definiteness of X does not guarantee the existence of a factor with the required sparsity. An error message results if the factorization is not possible. If the factorization is successful, \( R'*R \) agrees with X over its sparsity pattern.

\[ [R,p] = \text{cholinc}(X,'0') \]
with two output arguments, never produces an error message. If R exists, p is 0. If R does not exist, then p is a positive integer and R is an upper triangular matrix of size q-by-n where q = p - 1. In this latter case, the sparsity pattern of R is that of the q-by-n upper triangle of X. \( R'*R \) agrees with X over the sparsity pattern of its first q rows and first q columns.

R = cholinc(X,'inf') produces the Cholesky-Infinity factorization. This factorization is based on the Cholesky factorization, and additionally handles real positive semi-definite matrices. It may be useful for finding a solution to systems which arise in interior-point methods. When a zero pivot is encountered in the ordinary Cholesky factorization, the diagonal of the Cholesky-Infinity factor is set to \( \text{Inf} \) and the rest of that row is set to 0. This forces a 0 in the corresponding entry of the solution vector in the associated system of linear equations. In practice, X is assumed to be positive semi-definite so even negative pivots are replaced with a value of \( \text{Inf} \).

Remarks

The incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. A single 0 on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the rdiag option to replace a zero diagonal only
gets rid of the symptoms of the problem, but it does not solve it. The preconditioner may not be singular, but it probably is not useful, and a warning message is printed.

The Cholesky-Infinity factorization is meant to be used within interior-point methods. Otherwise, its use is not recommended.

**Examples**

**Example 1.**

Start with a symmetric positive definite matrix, \( S \).

\[
S = \text{delsq}(\text{numgrid}('C',15));
\]

\( S \) is the two-dimensional, five-point discrete negative Laplacian on the grid generated by \( \text{numgrid}('C',15) \).

Compute the Cholesky factorization and the incomplete Cholesky factorization of level 0 to compare the fill-in. Make \( S \) singular by zeroing out a diagonal entry and compute the (partial) incomplete Cholesky factorization of level 0.

\[
\begin{align*}
C &= \text{chol}(S); \\
R0 &= \text{cholinc}(S,'0'); \\
S2 &= S; \quad S2(101,101) = 0; \\
[R,p] &= \text{cholinc}(S2,'0');
\end{align*}
\]

Fill-in occurs within the bands of \( S \) in the complete Cholesky factor, but none in the incomplete Cholesky factor. The incomplete factorization of the singular \( S2 \) stopped at row \( p = 101 \) resulting in a 100-by-139 partial factor.

\[
\begin{align*}
D1 &= (R0'*R0).*\text{spones}(S) - S; \\
D2 &= (R'*R).*\text{spones}(S2) - S2;
\end{align*}
\]

\( D1 \) has elements of the order of \( \text{eps} \), showing that \( R0'*R0 \) agrees with \( S \) over its sparsity pattern. \( D2 \) has elements of the order of \( \text{eps} \) over its first 100 rows and first 100 columns, \( D2(1:100,:) \) and \( D2(:,1:100) \).
Example 2.

The first subplot below shows that \texttt{cholinc(S,0)}, the incomplete Cholesky factor with a drop tolerance of 0, is the same as the Cholesky factor of \texttt{S}. Increasing the drop tolerance increases the sparsity of the incomplete factors, as seen below.
Unfortunately, the sparser factors are poor approximations, as is seen by the plot of drop tolerance versus \( \text{norm}(R'\cdot R - S, 1) / \text{norm}(S, 1) \) in the next figure.

Example 3.

The Hilbert matrices have \((i,j)\) entries \(1/(i+j-1)\) and are theoretically positive definite:

\[
H_3 = \text{hilb}(3)
\]

\[
H_3 = \\
\begin{bmatrix}
1.0000 & 0.5000 & 0.3333 \\
0.5000 & 0.3333 & 0.2500 \\
0.3333 & 0.2500 & 0.2000
\end{bmatrix}
\]

\[
R_3 = \text{chol}(H_3)
\]

\[
R_3 = \\
\begin{bmatrix}
1.0000 & 0.5000 & 0.3333 \\
0 & 0.2887 & 0.2887 \\
0 & 0 & 0.0745
\end{bmatrix}
\]

In practice, the Cholesky factorization breaks down for larger matrices:

\[
H_{20} = \text{sparse(hilb(20))};
\]

\[
[R, p] = \text{chol}(H_{20});
\]

\[
p = 14
\]
For \texttt{hilb(20)}, the Cholesky factorization failed in the computation of row 14 because of a numerically zero pivot. You can use the Cholesky-Infinity factorization to avoid this error. When a zero pivot is encountered, \texttt{cholinc} places an Inf on the main diagonal, zeros out the rest of the row, and continues with the computation:

\begin{verbatim}
Rinf = cholinc(H20,'inf');
\end{verbatim}

In this case, all subsequent pivots are also too small, so the remainder of the upper triangular factor is:

\begin{verbatim}
full(Rinf(14:end,14:end))
\end{verbatim}

\begin{verbatim}
ans =
      Inf     0     0     0     0     0     0
       0     Inf     0     0     0     0     0
       0     0     Inf     0     0     0     0
       0     0     0     Inf     0     0     0
       0     0     0     0     Inf     0     0
       0     0     0     0     0     Inf     0
       0     0     0     0     0     0     Inf
\end{verbatim}

**Limitations**

\texttt{cholinc} works on square sparse matrices only. For \texttt{cholinc(X,'0')} and \texttt{cholinc(X,'inf')}, \texttt{X} must be real.

**Algorithm**

\texttt{R = cholinc(X,droptol)} is obtained from \([L,U] = luinc(X,options)\), where \texttt{options.droptol = droptol} and \texttt{options.thresh = 0}. The rows of the upper triangular \(U\) are scaled by the square root of the diagonal in that row, and this scaled factor becomes \(R\).

\texttt{R = cholinc(X,options)} is produced in a similar manner, except the \texttt{rdiag} option translates into the \texttt{udiag} option and the \texttt{milu} option takes the value of the \texttt{michol} option.

\texttt{R = cholinc(X,'0')} is based on the “KJI” variant of the Cholesky factorization. Updates are made only to positions which are nonzero in the upper triangle of \(X\).

\texttt{R = cholinc(X,'inf')} is based on the algorithm in Zhang [2].
See Also    chol, luinc, pcg

References

### cholupdate

**Purpose**

Rank 1 update to Cholesky factorization

**Syntax**

\[
\begin{align*}
R1 &= \text{cholupdate}(R,x) \\
R1 &= \text{cholupdate}(R,x,\,'+\') \\
R1 &= \text{cholupdate}(R,x,\,'-\') \\
[R1,p] &= \text{cholupdate}(R,x,\,'-\')
\end{align*}
\]

**Description**

- \( R1 = \text{cholupdate}(R,x) \) where \( R = \text{chol}(A) \) is the original Cholesky factorization of \( A \), returns the upper triangular Cholesky factor of \( A + x*x' \), where \( x \) is a column vector of appropriate length. `cholupdate` uses only the diagonal and upper triangle of \( R \). The lower triangle of \( R \) is ignored.

- \( R1 = \text{cholupdate}(R,x,\,'+\') \) is the same as \( R1 = \text{cholupdate}(R,x) \).

- \( R1 = \text{cholupdate}(R,x,\,'-\') \) returns the Cholesky factor of \( A - x*x' \). An error message reports when \( R \) is not a valid Cholesky factor or when the downdated matrix is not positive definite and so does not have a Cholesky factorization.

- \([R1,p] = \text{cholupdate}(R,x,\,'-\')\) will not return an error message. If \( p \) is 0, \( R1 \) is the Cholesky factor of \( A - x*x' \). If \( p \) is greater than 0, \( R1 \) is the Cholesky factor of the original \( A \). If \( p \) is 1, `cholupdate` failed because the downdated matrix is not positive definite. If \( p \) is 2, `cholupdate` failed because the upper triangle of \( R \) was not a valid Cholesky factor.

**Remarks**

`cholupdate` works only for full matrices.

**Example**

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

\[
A = \\
\begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 20
\end{pmatrix}
\]

\[
R = \text{chol}(A)
\]
\[ R = \]
\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 1 & 2 & 3 \\
0 & 0 & 1 & 3 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
\[ x = [0 \ 0 \ 0 \ 1]' \]

This is called a rank one update to \( A \) since \( \text{rank}(x^*x') \) is 1:

\[ A + x^*x' \]
\[ \text{ans} = \]
\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 21
\end{bmatrix}
\]

Instead of computing the Cholesky factor with \( R1 = \text{chol}(A + x^*x') \), we can use \text{cholupdate}:

\[ R1 = \text{cholupdate}(R,x) \]
\[ R1 = \]
\[
\begin{bmatrix}
1.0000 & 1.0000 & 1.0000 & 1.0000 \\
0 & 1.0000 & 2.0000 & 3.0000 \\
0 & 0 & 1.0000 & 3.0000 \\
0 & 0 & 0 & 1.4142
\end{bmatrix}
\]

Next destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element of \( A \). The downdated matrix is:

\[ A - x^*x' \]
\[ \text{ans} = \]
\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
1 & 3 & 6 & 10 \\
1 & 4 & 10 & 19
\end{bmatrix}
\]
cholupdate

Compare chol with cholupdate:

```matlab
R1 = chol(A-x*x')
??? Error using ==> chol
Matrix must be positive definite.
```

```matlab
R1 = cholupdate(R,x,'-')
??? Error using ==> cholupdate
Downdated matrix must be positive definite.
```

However, subtracting 0.5 from the last element of A produces a positive definite matrix, and we can use cholupdate to compute its Cholesky factor:

```matlab
x = [0 0 0 1/sqrt(2)]';
R1 = cholupdate(R,x,'-')
```

```matlab
R1 =
1.0000    1.0000    1.0000    1.0000
0    1.0000    2.0000    3.0000
0    0    1.0000    3.0000
0    0    0    0.7071
```

**Algorithm**

cholupdate uses the algorithms from the LINPACK subroutines ZCHUD and ZCHDD. cholupdate is useful since computing the new Cholesky factor from scratch is an $O(N^3)$ algorithm, while simply updating the existing factor in this way is an $O(N^2)$ algorithm.

**See Also**

chol, qrupdate

**References**

Purpose
Shift array circularly

Syntax
B = circshift(A,shiftsize)

Description
B = circshift(A,shiftsize) circularly shifts the values in the array, A, by
shiftsize elements. shiftsize is a vector of integer scalars where the n-th
element specifies the shift amount for the n-th dimension of array A. If an
element in shiftsize is positive, the values of A are shifted down (or to the
right). If it is negative, the values of A are shifted up (or to the left). If it is 0,
the values in that dimension are not shifted.

Example
Circularly shift first dimension values down by 1.
A = [ 1 2 3; 4 5 6; 7 8 9]
A =
 1    2    3
 4    5    6
 7    8    9

B = circshift(A,1)
B =
 7    8    9
 1    2    3
 4    5    6

Circularly shift first dimension values down by 1 and second dimension values
to the left by 1.
B = circshift(A,[1 -1]);
B =
 8    9    7
 2    3    1
 5    6    4

See Also
fftshift, shiftdim
# cla

## Purpose
Clear current axes

## Syntax
- `cla`
- `cla reset`

## Description
`cla` deletes from the current axes all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to `on`).

`cla reset` deletes from the current axes all graphics objects regardless of the setting of their `HandleVisibility` property and resets all axes properties, except `Position` and `Units`, to their default values.

## Remarks
The `cla` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `cla` deletes only those objects whose `HandleVisibility` property is set to `on`.

## See Also
- `clf`, `hold`, `newplot`, `reset`
- “Axes Operations” for related functions
Purpose
Contour plot elevation labels

Syntax
clabel(C,h)
clabel(C,h,v)
clabel(C,h,'manual')

clabel(C)
clabel(C,v)
clabel(C,'manual')

text_handles = clabel(...)
clabel(...,'PropertyName',propertyvalue,...)
clabel(...'LabelSpacing',points)

Description
The clabel function adds height labels to a two-dimensional contour plot.

clabel(C,h) rotates the labels and inserts them in the contour lines. The function inserts only those labels that fit within the contour, depending on the size of the contour.

clabel(C,h,v) creates labels only for those contour levels given in vector v, then rotates the labels and inserts them in the contour lines.

clabel(C,h,'manual') places contour labels at locations you select with a mouse. Press the left mouse button (the mouse button on a single-button mouse) or the space bar to label a contour at the closest location beneath the center of the cursor. Press the Return key while the cursor is within the figure window to terminate labeling. The labels are rotated and inserted in the contour lines.

clabel(C) adds labels to the current contour plot using the contour array C output from contour. The function labels all contours displayed and randomly selects label positions.

clabel(C,v) labels only those contour levels given in vector v.

clabel(C,'manual') places contour labels at locations you select with a mouse.
text_handles = clabel(...) returns the handles of text objects created by clabel. The UserData properties of the text objects contain the contour values displayed. If you call clabel without the h argument, text_handles also contains the handles of line objects used to create the '+' symbols.

clabel(...,'PropertyName',propertyvalue,...) enables you to specify text object property/value pairs for the label strings. (See text properties.)

clabel(...,'LabelSpacing',points) specifies the spacing between labels on the same contour line, in units of points (72 points equal one inch).

Remarks
When the syntax includes the argument h, this function rotates the labels and inserts them in the contour lines (see Examples). Otherwise, the labels are displayed upright and a '+' indicates which contour line the label is annotating.

Examples
Generate, draw, and label a simple contour plot.

```matlab
[x,y] = meshgrid(-2:.2:2);
z = x.^exp(-x.^2-y.^2);
[C,h] = contour(x,y,z);
clabel(C,h);
```
Label a contour plot with label spacing set to 72 points (one inch).

```matlab
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'LabelSpacing',72)
```
Label a contour plot with 15 point red text.

```matlab
[x,y,z] = peaks;
[C,h] = contour(x,y,z);
clabel(C,h,'FontSize',15,'Color','r','Rotation',0)
```
Label a contour plot with upright text and '+' symbols indicating which contour line each label annotates.

```
[x,y,z] = peaks;
C = contour(x,y,z);
clabel(C)
```
See Also

contour, contourc, contourf

“Annotating Plots” for related functions

Drawing Text in a Box for an example that illustrates the use of contour labels
class

**Purpose**

Create object or return class of object

**Syntax**

```matlab
str = class(object)
obj = class(s,'class_name')
obj = class(s,'class_name',parent1,parent2...)
obj = class(struct([]),'class_name',parent1,parent2...)
```

**Description**

`str = class(object)` returns a string specifying the class of `object`. The following table lists the object class names that can be returned. All except the last one are MATLAB classes.

<table>
<thead>
<tr>
<th><code>class_name</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>logical</code></td>
<td>Logical array of true and false values</td>
</tr>
<tr>
<td><code>char</code></td>
<td>Character array</td>
</tr>
<tr>
<td><code>int8</code></td>
<td>8-bit signed integer array</td>
</tr>
<tr>
<td><code>uint8</code></td>
<td>8-bit unsigned integer array</td>
</tr>
<tr>
<td><code>int16</code></td>
<td>16-bit signed integer array</td>
</tr>
<tr>
<td><code>uint16</code></td>
<td>16-bit unsigned integer array</td>
</tr>
<tr>
<td><code>int32</code></td>
<td>32-bit signed integer array</td>
</tr>
<tr>
<td><code>uint32</code></td>
<td>32-bit unsigned integer array</td>
</tr>
<tr>
<td><code>int64</code></td>
<td>64-bit signed integer array</td>
</tr>
<tr>
<td><code>uint64</code></td>
<td>64-bit unsigned integer array</td>
</tr>
<tr>
<td><code>single</code></td>
<td>Single-precision floating-point number array</td>
</tr>
<tr>
<td><code>double</code></td>
<td>Double-precision floating-point number array</td>
</tr>
<tr>
<td><code>cell</code></td>
<td>Cell array</td>
</tr>
<tr>
<td><code>struct</code></td>
<td>Structure array</td>
</tr>
<tr>
<td><code>function handle</code></td>
<td>Array of values for calling functions indirectly</td>
</tr>
<tr>
<td><code>'class_name'</code></td>
<td>Custom MATLAB object class or Java class</td>
</tr>
</tbody>
</table>

`obj = class(s,'class_name')` creates an object of MATLAB class `'class_name'` using structure `s` as a template. This syntax is valid only in a
function named `class_name.m` in a directory named `@class_name` (where 'class_name' is the same as the string passed in to `class`).

`obj = class(s,'class_name',parent1,parent2,...)` creates an object of MATLAB class 'class_name' that inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. Structure `s` is used as a template for the object.

`obj = class(struct([]),'class_name',parent1,parent2,...)` creates an object of MATLAB class 'class_name' that inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. Specifying the empty structure `struct([])` as the first argument ensures that the object created contains no fields other than those that are inherited from the parent objects.

**Examples**

To return in `nameStr` the name of the class of Java object `j`,

```matlab
nameStr = class(j)
```

To create a user-defined MATLAB object of class `polynom`,

```matlab
p = class(p,'polynom')
```

**See Also**

`inferiorto`, `isa`, `superiorto`

The “MATLAB Classes and Objects” and the “Calling Java from MATLAB” chapters in MATLAB Programming and Data Types documentation.
### clc

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Clear Command Window</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graphical Interface</strong></td>
<td>As an alternative to the clc function, use <strong>Clear Command Window</strong> in the MATLAB desktop <strong>Edit</strong> menu.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>clc</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>clc clears all input and output from the Command Window display, giving you a “clean screen.” After using clc, you cannot use the scroll bar to see the history of functions, but you still can use the up arrow to recall statements from the command history.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>Use clc in an M-file to always display output in the same starting position on the screen.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>clear, clf, close, home</td>
</tr>
</tbody>
</table>
clear

**Purpose**
Remove items from workspace, freeing up system memory

**Graphical Interface**
As an alternative to the `clear` function, use **Clear Workspace** in the MATLAB desktop **Edit** menu.

**Syntax**
```
clear
clear name
clear name1 name2 name3 ...
clear global name
clear -regexp expr1 expr2 ...
clear global -regexp expr1 expr2 ...
clear keyword
clear('name1','name2','name3',...)```

**Description**
clear removes all variables from the workspace. This frees up system memory.

clear name removes just the M-file or MEX-file function or variable name from the workspace. You can use wildcards (*) to remove items selectively. For example, clear my* removes any variables whose names begin with the string my. It removes debugging breakpoints in M-files and reinitializes persistent variables, since the breakpoints for a function and persistent variables are cleared whenever the M-file is changed or cleared. If name is global, it is removed from the current workspace, but left accessible to any functions declaring it global. If name has been locked by mlock, it remains in memory.

Use a partial path to distinguish between different overloaded versions of a function. For example, clear polynom/display clears only the display method for polynom objects, leaving any other implementations in memory.

clear name1 name2 name3 ... removes name1, name2, and name3 from the workspace.

clear global name removes the global variable name. If name is global, clear name removes name from the current workspace, but leaves it accessible to any functions declaring it global. Use clear global name to completely remove a global variable.

clear -regexp expr1 expr2 ... clears all variables that match any of the regular expressions expr1, expr2, etc. This option only clears variables.
clear global -regexp expr1 expr2 ... clears all global variables that match any of the regular expressions expr1, expr2, etc.

clear keyword clears the items indicated by keyword.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Items Cleared</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Removes all variables, functions, and MEX-files from memory, leaving the workspace empty. Using clear all removes debugging breakpoints in M-files and reinitializes persistent variables, since the breakpoints for a function and persistent variables are cleared whenever the M-file is changed or cleared. When issued from the Command Window prompt, also removes the Java packages import list.</td>
</tr>
<tr>
<td>classes</td>
<td>The same as clear all, but also clears MATLAB class definitions. If any objects exist outside the workspace (for example, in user data or persistent variables in a locked M-file), a warning is issued and the class definition is not cleared. Issue a clear classes function if the number or names of fields in a class are changed.</td>
</tr>
<tr>
<td>functions</td>
<td>Clears all the currently compiled M-functions and MEX-functions from memory. Using clear function removes debugging breakpoints in the function M-file and reinitializes persistent variables, since the breakpoints for a function and persistent variables are cleared whenever the M-file is changed or cleared.</td>
</tr>
<tr>
<td>global</td>
<td>Clears all global variables from the workspace.</td>
</tr>
<tr>
<td>import</td>
<td>Removes the Java packages import list. It can only be issued from the Command Window prompt. It cannot be used in a function.</td>
</tr>
</tbody>
</table>
clear

<table>
<thead>
<tr>
<th>java</th>
<th>The same as clear all, but also clears the definitions of all Java classes defined by files on the Java dynamic class path (see “The Java Class Path” in the External Interfaces documentation). If any java objects exist outside the workspace (for example, in user data or persistent variables in a locked M-file), a warning is issued and the Java class definition is not cleared. Issue a clear java command after modifying any files on the Java dynamic class path.</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>Clears all variables from the workspace.</td>
</tr>
</tbody>
</table>

clear(‘name1’, ‘name2’, ‘name3’, ...) is the function form of the syntax. Use this form when the variable name or function name is stored in a string.

Remarks

When you use clear in a function, it has the following effect on items in your function and base workspaces:

- **clear name**—If name is the name of a function, the function is cleared in both the function workspace and in your base workspace.
- **clear functions**—All functions are cleared in both the function workspace and in your base workspace.
- **clear global**—All global variables are cleared in both the function workspace and in your base workspace.
- **clear all**—All functions, global variables, and classes are cleared in both the function workspace and in your base workspace.

Limitations

clear does not affect the amount of memory allocated to the MATLAB process under UNIX.

The clear function does not clear Simulink models. Use close instead.

Examples

Given a workspace containing the following variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>3x4</td>
<td>1200</td>
<td>cell array</td>
</tr>
<tr>
<td>frame</td>
<td>1x1</td>
<td></td>
<td>java.awt.Frame</td>
</tr>
<tr>
<td>gbl1</td>
<td>1x1</td>
<td>8</td>
<td>double array (global)</td>
</tr>
</tbody>
</table>
you can clear a single variable, `xint`, by typing
```
clear xint
```
To clear all global variables, type
```
clear global
```
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>3x4</td>
<td>1200</td>
<td>cell array</td>
</tr>
<tr>
<td>frame</td>
<td>1x1</td>
<td></td>
<td>java.awt.Frame</td>
</tr>
</tbody>
</table>
```
Using regular expressions, clear those variables with names that begin with `Mon`, `Tue`, or `Wed`:
```
clear('-regexp', '^Mon|^Tue|^Wed');
```
To clear all compiled M- and MEX-functions from memory, type `clear` functions. In the case shown below, `clear functions` was unable to clear one M-file function from memory, `testfun`, because the function is locked.
```
clear functions           % Attempt to clear all functions.
inmem ans =
    'testfun'             % One M-file function remains in memory.
mislocked testfun ans =
    1                    % This function is locked in memory.
```
Once you unlock the function from memory, you can clear it.
```
munlock testfun
clear functions
```
```
inmem ans =
    Empty cell array: 0-by-1
```
clear

See Also  clc, close, import, inmem, load, mlock, munlock, pack, persistent, save, who, whos, workspace
**Purpose**  
Clear current figure window

**Syntax**

```matlab
clf
clf('reset')
figure_handle = clf(...)
```

**Description**

`clf` deletes from the current figure all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to on).

`clf('reset')` deletes from the current figure all graphics objects regardless of the setting of their `HandleVisibility` property and resets all figure properties except `Position`, `Units`, `PaperPosition`, and `PaperUnits` to their default values.

`figure_handle = clf(...)` return the handle of the figure. This is useful when the figure `IntegerHandle` property is off since the noninteger handle becomes invalid when the reset option is used (i.e., `IntegerHandle` is reset to on, which is the default).

**Remarks**

The `clf` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `clf` deletes only those objects whose `HandleVisibility` property is set to on.

**See Also**

`cla`, `clc`, `hold`, `reset`

“Figure Windows” for related functions
clipboard

**Purpose**
Copy and paste strings to and from the system clipboard

**Syntax**
```matlab
clipboard('copy', data)
str = clipboard('paste')
data = clipboard('pastespecial')
```

**Description**
- `clipboard('copy', data)` sets the clipboard contents to `data`. If `data` is not a character array, the clipboard uses `mat2str` to convert it to a string.
- `str = clipboard('paste')` returns the current contents of the clipboard as a string or as an empty string (`''`), if the current clipboard contents cannot be converted to a string.
- `data = clipboard('pastespecial')` returns the current contents of the clipboard as an array using `uiimport`.

**Note**
Requires an active X display on UNIX, and Java elsewhere.

**See Also**
`load`, `uiimport`
Purpose
Current time as a date vector

Syntax
c = clock

Description
c = clock returns a 6-element date vector containing the current date and
time in decimal form:

\[ c = [\text{year} \ \text{month} \ \text{day} \ \text{hour} \ \text{minute} \ \text{seconds}] \]

The first five elements are integers. The seconds element is accurate to several
digits beyond the decimal point. The statement \text{fix(clock)} rounds to integer
display format.

See Also
\text{cputime}, \text{datenum}, \text{datevec}, \text{etime}, \text{tic}, \text{toc}
close

Purpose
Delete specified figure

Syntax
close
close(h)
close name
close all
close all hidden
status = close(...)

Description
close deletes the current figure or the specified figure(s). It optionally returns
the status of the close operation.

close deletes the current figure (equivalent to close(gcf)).

close(h) deletes the figure identified by h. If h is a vector or matrix, close
deletes all figures identified by h.

close name deletes the figure with the specified name.

close all deletes all figures whose handles are not hidden.

close all hidden deletes all figures including those with hidden handles.

status = close(...) returns 1 if the specified windows have been deleted
and 0 otherwise.

Remarks
The close function works by evaluating the specified figure's CloseRequestFcn
property with the statement

    eval(get(h,'CloseRequestFcn'))

The default CloseRequestFcn, closereq, deletes the current figure using
delete(get(0,'CurrentFigure')). If you specify multiple figure handles,
close executes each figure's CloseRequestFcn in turn. If MATLAB encounters
an error that terminates the execution of a CloseRequestFcn, the figure is not
deleted. Note that using your computer's window manager (i.e., the Close
menu item) also calls the figure's CloseRequestFcn.

If a figure's handle is hidden (i.e., the figure's HandleVisibility property is set
to callback or off and the root ShowHiddenHandles property is set on), you

2-372
must specify the hidden option when trying to access a figure using the all option.

To delete all figures unconditionally, use the statements

```matlab
set(0,'ShowHiddenHandles','on')
delete(get(0,'Children'))
```

The delete function does not execute the figure’s CloseRequestFcn; it simply deletes the specified figure.

The figure CloseRequestFcn allows you to either delay or abort the closing of a figure once the close function has been issued. For example, you can display a dialog box to see if the user really wants to delete the figure or save and clean up before closing.

**See Also**

dele, figure, gcf

The figure HandleVisibility property

The root ShowHiddenHandles property

“Figure Windows” for related functions
**close (avifile)**

**Purpose**
Close Audio/Video Interleaved (AVI) file

**Syntax**
`aviobj = close(aviobj)`

**Description**
`aviobj = close(aviobj)` finishes writing and closes the AVI file associated with `aviobj`, which is an AVI file object created using the `avifile` function.

**See Also**
avifile, addframe, movie2avi
Close connection with FTP server

close(f)

close(f) closes the connection with the FTP server, represented by object f, which was created using ftp. Be sure to use close after completing work on the server. If you do not run close, the connection will be terminated automatically either because of the server's time-out feature or when you exit MATLAB.

Connect to The MathWorks FTP server and then disconnect.

```matlab
tmw = ftp('ftp.mathworks.com');
close(tmw)
ans =
disconnected
```

See Also

ftp
Purpose: Default figure close request function

Syntax: closereq

Description: closereq deletes the current figure.

See Also: The figure CloseRequestFcn property

“Figure Windows” for related functions
## cmopts

### Purpose
Get name of source control system

### Graphical Interface
As an alternative to `cmopts`, use preferences. Select **File -> Preferences** in the MATLAB desktop, and then select **General -> Source Control**.

### Syntax
`cmopts`

### Description
`cmopts` returns the name of the source control system you selected using preferences, which is one of the following:

- `clearcase`
- `customverctrl`
- `pvcs`
- `rcs`
- `sourcesafe`

If you have not selected a source control system, `cmopts` returns `none`.

### Specifying a Source Control System
To specify the source control system:

1. From the MATLAB Editor window or from a Simulink or Stateflow model window, select **File -> Preferences**. The **Preferences** dialog box opens.
2. In the left pane, click the + for **General**, and then select **Source Control**. The currently selected system is shown.
3. Select the system you want to use from the **Source control system** list.
4. Click **OK**.

For more information, see source control preferences.

### Examples
Type `cmopts` and MATLAB returns `rcs`, meaning the source control system specified in preferences is RCS.

### See Also
`checkin`, `checkout`, `customverctrl`
**Purpose**

Column approximate minimum degree permutation

**Syntax**

```matlab
p = colamd(S)
p = colamd(S,knobs)
[p,stats] = colamd(S)
[p,stats] = colamd(S,knobs)
```

**Description**

`p = colamd(S)` returns the column approximate minimum degree permutation vector for the sparse matrix `S`. For a non-symmetric matrix `S`, `S(:,p)` tends to have sparser LU factors than `S`. The Cholesky factorization of `S(:,p)' * S(:,p)` also tends to be sparser than that of `S'*S`.

`knobs` is a two-element vector. If `S` is `m`-by-`n`, then rows with more than `(knobs(1))*n` entries are ignored. Columns with more than `(knobs(2))*m` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs(1) = knobs(2) = spparms('wh_frac')`.

`stats` is an optional vector that provides data about the ordering and the validity of the matrix `S`.

- `stats(1)` Number of dense or empty rows ignored by `colamd`
- `stats(2)` Number of dense or empty columns ignored by `colamd`
- `stats(3)` Number of garbage collections performed on the internal data structure used by `colamd` (roughly of size `2.2*nnz(S) + 4*m + 7*n` integers)
- `stats(4)` 0 if the matrix is valid, or 1 if invalid
- `stats(5)` Rightmost column index that is unsorted or contains duplicate entries, or 0 if no such column exists
- `stats(6)` Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or 0 if no such row index exists
- `stats(7)` Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `colamd`. For this reason, `colamd` verifies that `S` is valid:
• If a row index appears two or more times in the same column, \texttt{colamd} 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{colamd} sorts each column of its internal copy of the matrix \texttt{S} (but does not repair the input matrix \texttt{S}), continues processing, and provides information about the out-of-order entries in \texttt{stats(4:7)}.

• If \texttt{S} is invalid in any other way, \texttt{colamd} cannot continue. It prints an error message, and returns no output arguments (\texttt{p} or \texttt{stats}).

The ordering is followed by a column elimination tree post-ordering.

\textbf{Note} \texttt{colamd} tends to be faster than \texttt{colmmd} and tends to return a better ordering.

\textbf{See Also}\n
\texttt{colmmd}, \texttt{colperm}, \texttt{spparms}, \texttt{symamd}, \texttt{symmmd}, \texttt{symrcm}

\textbf{References}\n
[1] The authors of the code for \texttt{colamd} are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. Sparse Matrix Algorithms Research at the University of Florida: http://www.cise.ufl.edu/research/sparse/
Purpose
Sparse column minimum degree permutation

Syntax
\[ p = \text{colmmd}(S) \]

Description
\[ p = \text{colmmd}(S) \] returns the column minimum degree permutation vector for the sparse matrix \( S \). For a nonsymmetric matrix \( S \), this is a column permutation \( p \) such that \( S(:,p) \) tends to have sparser LU factors than \( S \).

The \text{colmmd} permutation is automatically used by \( \backslash \) and \( / \) for the solution of nonsymmetric and symmetric indefinite sparse linear systems.

Use \text{spparms} to change some options and parameters associated with heuristics in the algorithm.

Algorithm
The minimum degree algorithm for symmetric matrices is described in the review paper by George and Liu [1]. For nonsymmetric matrices, the MATLAB minimum degree algorithm is new and is described in the paper by Gilbert, Moler, and Schreiber [2]. It is roughly like symmetric minimum degree for \( A' \cdot A \), but does not actually form \( A' \cdot A \).

Each stage of the algorithm chooses a vertex in the graph of \( A' \cdot A \) of lowest degree (that is, a column of \( A \) having nonzero elements in common with the fewest other columns), eliminates that vertex, and updates the remainder of the graph by adding fill (that is, merging rows). If the input matrix \( S \) is of size \( m \)-by-\( n \), the columns are all eliminated and the permutation is complete after \( n \) stages. To speed up the process, several heuristics are used to carry out multiple stages simultaneously.

Examples
The Harwell-Boeing collection of sparse matrices and the MATLAB demos directory include a test matrix WEST0479. It is a matrix of order 479 resulting from a model due to Westerberg of an eight-stage chemical distillation column. The spy plot shows evidence of the eight stages. The colmmd ordering scrambles this structure.

\begin{verbatim}
load west0479
A = west0479;
p = colmmd(A);
spy(A)
spy(A(:,p))
\end{verbatim}
Comparing the spy plot of the LU factorization of the original matrix with that of the reordered matrix shows that minimum degree reduces the time and storage requirements by better than a factor of 2.8. The nonzero counts are 16777 and 5904, respectively.

```matlab
spy(lu(A))
spy(lu(A(:,p)))
```
See Also

colamd, colperm, lu, ssparms, symamd, symmmd, symrcm

The arithmetic operator \n
References


Purpose
Display colorbar showing the color scale

Syntax

```
colorbar
colorbar(...,'peer',axes_handle)
colorbar(axes_handle)
colorbar('location')
colorbar(...,'PropertyName',propertyvalue)
cbar_axes = colorbar(...)
```

Description
The `colorbar` function displays the current colormap in the current figure and resizes the current axes to accommodate the colorbar.

colorbar updates the most recently created colorbar or, when the current axes does not have a colorbar, colorbar adds a new vertical colorbar.

colorbar(...,'peer',axes_handle) creates a colorbar associated with the axes axes_handle instead of the current axes.

colorbar(axes_handle) adds the colorbar to the axes axes_handle in the default (right) orientation.

colorbar(...,'location') adds a colorbar in the specified orientation with respect to the axes. Possible values for location are

- North — inside plot box near top
- South — inside bottom
- East — inside right
- West — inside left
- NorthOutside — outside plotbox near top
- SouthOutside — outside bottom
- EastOutside — outside right
- WestOutside — outside left

colorbar(...,'PropertyName',propertyvalue) specifies property names and values for the axes object used to create the colorbar. See axes properties for a description of the properties you can set.
colorbar

cbar_axes = colorbar(...) returns a handle to the colorbar, which is an axes graphics object that contains one additional property, Location.

Remarks
You can use colorbar with 2-D and 3-D plots.

Examples
Display a colorbar beside the axes and use descriptive text strings as y-tick labels.

```
surf(peaks(30))
colorbar('YTickLabel',...)
    {'Freezing','Cold','Cool','Neutral','Warm','Hot','Burning'})
```

See Also
colormap
“Color Operations” for related functions
Purpose

Set default property values to display different color schemes

Syntax

colordef white

Syntax

colordef black

colordef none

colordef(fig,color_option)

h = colordef('new',color_option)

Description

colordef enables you to select either a white or black background for graphics display. It sets axis lines and labels to show up against the background color.

colordef white sets the axis background color to white, the axis lines and labels to black, and the figure background color to light gray.

colordef black sets the axis background color to black, the axis lines and labels to white, and the figure background color to dark gray.

colordef none sets the figure coloring to that used by MATLAB Version 4 (essentially a black background).

colordef(fig,color_option) sets the color scheme of the figure identified by the handle fig to the color option 'white', 'black', or 'none'.

h = colordef('new',color_option) returns the handle to a new figure created with the specified color options (i.e., 'white', 'black', or 'none').

Remarks

colordef affects only subsequently drawn figures, not those currently on the display. This is because colordef works by setting default property values (on the root or figure level). You can list the currently set default values on the root level with the statement

get(0,'defaults')

You can remove all default values using the reset command:

reset(0)

See the get and reset references pages for more information.

See Also

whitebg
“Color Operations” for related functions
Purpose
Set and get the current colormap

Syntax
```
colormap(map)
colormap('default')
cmap = colormap
```

Description
A colormap is an \( m \)-by-3 matrix of real numbers between 0.0 and 1.0. Each row is an RGB vector that defines one color. The \( k \)th row of the colormap defines the \( k \)th color, where \( \text{map}(k,:) = [r(k) \ g(k) \ b(k)] \) specifies the intensity of red, green, and blue.

\( \text{colormap}(\text{map}) \) sets the colormap to the matrix \( \text{map} \). If any values in \( \text{map} \) are outside the interval \([0 \ 1]\), MATLAB returns the error \text{Colormap must have values in [0,1]}.

\( \text{colormap('default')} \) sets the current colormap to the default colormap.

\( \text{cmap} = \text{colormap} \); retrieves the current colormap. The values returned are in the interval \([0 \ 1]\).

Specifying Colormaps
M-files in the \text{color} directory generate a number of colormaps. Each M-file accepts the colormap size as an argument. For example,

```
colormap(hsv(128))
```
creates an \text{hsv} colormap with 128 colors. If you do not specify a size, MATLAB creates a colormap the same size as the current colormap.

Supported Colormaps
MATLAB supports a number of colormaps.

- \text{autumn} varies smoothly from red, through orange, to yellow.
- \text{bone} is a grayscale colormap with a higher value for the blue component. This colormap is useful for adding an “electronic” look to grayscale images.
- \text{colorcube} contains as many regularly spaced colors in RGB colorspace as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
colormap

- **cool** consists of colors that are shades of cyan and magenta. It varies smoothly from cyan to magenta.
- **copper** varies smoothly from black to bright copper.
- **flag** consists of the colors red, white, blue, and black. This colormap completely changes color with each index increment.
- **gray** returns a linear grayscale colormap.
- **hot** varies smoothly from black through shades of red, orange, and yellow, to white.
- **hsv** varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for displaying periodic functions. \( \text{hsv}(m) \) is the same as \( \text{hsv2rgb}([h \ \text{ones}(m,2)]) \) where \( h \) is the linear ramp, \( h = (0:m-1)' / m \).
- **jet** ranges from blue to red, and passes through the colors cyan, yellow, and orange. It is a variation of the hsv colormap. The jet colormap is associated with an astrophysical fluid jet simulation from the National Center for Supercomputer Applications. See the “Examples” section.
- **lines** produces a colormap of colors specified by the axes ColorOrder property and a shade of gray.
- **pink** contains pastel shades of pink. The pink colormap provides sepia tone colorization of grayscale photographs.
- **prism** repeats the six colors red, orange, yellow, green, blue, and violet.
- **spring** consists of colors that are shades of magenta and yellow.
- **summer** consists of colors that are shades of green and yellow.
- **white** is an all white monochrome colormap.
- **winter** consists of colors that are shades of blue and green.

**Examples**

The images and colormaps demo, imagedemo, provides an introduction to colormaps. Select **Color Spiral** from the menu. This uses the pcolor function to display a 16-by-16 matrix whose elements vary from 0 to 255 in a rectilinear spiral. The hsv colormap starts with red in the center, then passes through yellow, green, cyan, blue, and magenta before returning to red at the outside end of the spiral. Selecting **Colormap Menu** gives access to a number of other colormaps.
The `rgbplot` function plots colormap values. Try `rgbplot(hsv)`, `rgbplot(gray)`, and `rgbplot(hot)`.

The following commands display the `flujet` data using the `jet` colormap.

```matlab
load flujet
image(X)
colormap(jet)
```

The demos directory contains a CAT scan image of a human spine. To view the image, type the following commands:

```matlab
load spine
image(X)
colormap bone
```
Algorithm

Each figure has its own Colormap property. colormap is an M-file that sets and gets this property.

See Also

brighten, caxis, colormapeditor, colorbar, contrast, hsv2rgb, pcolor, rgb2hsv, rgbplot

The Colormap property of figure graphics objects

“Color Operations” for related functions

Coloring Mesh and Surface Plots for more information about colormaps and other coloring methods
Purpose
Start colormap editor

Syntax
colormapeditor

Description
colormapeditor displays the current figure’s colormap as a strip of rectangular cells in the colormap editor. Node pointers are colored cells below the colormap strip that indicate points in the colormap where the rate of the variation of R, G, and B values changes. You can also work in the HSV colorspace by setting the Interpolating Colorspace selector to HSV.

You can also start the colormap editor by selecting Colormap from the Edit menu.

Node Pointer Operations
You can select and move node pointers to change a range of colors in the colormap. The color of a node pointer remains constant as you move it, but the colormap changes by linearly interpolating the RGB values between nodes.

Change the color at a node by double-clicking the node pointer. MATLAB displays a color picker from which you can select a new color. After you select a new color at a node, MATLAB reinterpolates the colors in between nodes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>How to Perform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a node</td>
<td>Click below the corresponding cell in the colormap strip.</td>
</tr>
<tr>
<td>Select a node</td>
<td>Left-click the node.</td>
</tr>
<tr>
<td>Select multiple nodes</td>
<td>Adjacent: left-click first node, Shift+click the last node.</td>
</tr>
<tr>
<td></td>
<td>Nonadjacent: left-click first node, Ctrl+click subsequent nodes.</td>
</tr>
<tr>
<td>Move a node</td>
<td>Select and drag with the mouse or select and use the left and right arrow keys.</td>
</tr>
</tbody>
</table>
Current Color Info
When you put the mouse over a color cell or node pointer, the colormap editor displays the following information about that colormap element:

- The element’s index in the colormap
- The value from the graphics object color data that is mapped to the node’s color (i.e., data from the CData property of any image, patch, or surface objects in the figure)
- The color’s RGB and HSV color value
Interpolating Colorspace

The colorspace determines what values are used to calculate the colors of cells between nodes. For example, in the RGB colorspace, internode colors are calculated by linearly interpolating the red, green, and blue intensity values from one node to the next. Switching to the HSV colorspace causes the colormap editor to recalculate the colors between nodes using the hue, saturation, and value components of the color definition.

Note that when you switch from one colorspace to another, the color editor preserves the number, color, and location of the node pointers, which can cause the colormap to change.

Interpolating in HSV: Since hue is conceptually mapped about a color circle, the interpolation between hue values can be ambiguous. To minimize this ambiguity, the interpolation uses the shortest distance around the circle. For example, interpolating between two nodes, one with hue of 2 (slightly orange red) and another with a hue of 356 (slightly magenta red), does not result in hues 3, 4, ..., 353, 354, 355 (orange/red-yellow-green-cyan-blue-magenta/red).
Taking the shortest distance around the circle gives 357,358,1,2 (orange/red-red-magenta/red).

**Color Data Min and Max**

The **Color Data Min** and **Color Data Max** text fields enable you to specify values for the axes **CLim** property. These values change the mapping of object color data (the **CData** property of images, patches, and surfaces) to the colormap. See Axes Color Limits — the Clim Property for discussion and examples of how to use this property.

**Examples**

This example modifies a default MATLAB colormap so that ranges of data values are displayed in specific ranges of color. The graph is a slice plane illustrating a cross section of fluid flow through a jet nozzle. See the slice reference page for more information on this type of graph.

**Example Objectives**

The objectives are as follows:

- Regions of flow from left to right (positive data) are mapped to colors from yellow through orange to dark red. Yellow is slowest and dark red is the fastest moving fluid.
- Regions that have a speed close to zero are colored green.
- Regions where the fluid is actually moving right to left (negative data) are shades of blue (darker blue is faster).

The following picture shows the desired coloring of the slice plane. The colorbar shows the data to color mapping.
Running the Example

**Note** If you are viewing this documentation in the MATLAB help browser, you can display the graph used in this example by running this M-file from the MATLAB editor (select Run from the Debug menu).

Initially, the default colormap *(jet)* colored the slice plane, as illustrated in the following picture. Note that this example uses a colormap that is 48 elements to display wider bands of color (the default is 64 elements).
Start the colormap editor using the `colormapeditor` command. The colormap editor displays the current figure’s colormap, as shown in the following picture.
Since we want the regions of left-to-right flow (positive speed) to range from yellow to dark red, we can delete the cyan node pointer. To do this, first select it by clicking with the left mouse button and press **Delete**. The colormap now looks like this.

**2** Since we want the regions of left-to-right flow (positive speed) to range from yellow to dark red, we can delete the cyan node pointer. To do this, first select it by clicking with the left mouse button and press **Delete**. The colormap now looks like this.
The **Immediate Apply** box is checked, so the graph displays the results of the changes made to the colormap.
We want the fluid speed values around zero to stand out, so we need to find the color cell where the negative-to-positive transition occurs. Dragging the cursor over the color strip enables you to read the data values in the **Current Color Info** panel.

In this case, cell 10 is the first positive value, so we click below that cell and create a node pointer. Double-clicking the node pointer displays the color picker. Set the color of this node to green.
The graph continues to update to the modified colormap.
4 In the current state, the colormap colors are interpolated from the green node to the yellowish node about 20 cells away. We actually want only the single cell that is centered around zero to be colored green. To limit the color green to one cell, move the blue and yellow node pointers next to the green pointer.

5 Before making further adjustments to the colormap, we need to move the green cell so that it is centered around zero. Use the colorbar to locate the green cell.
To recenter the green cell around zero, select the blue, green, and yellow node pointers (left-click blue, Shift+click yellow) and move them as a group using the left arrow key. Watch the colorbar in the figure window to see when the green color is centered around zero.

Note that green cell is not centered around zero.
The slice plane now has the desired range of colors for negative, zero, and positive data.
6 Increase the orange-red coloring in the slice by moving the red node pointer toward the yellow node.
7 Darken the endpoints to bring out more detail in the extremes of the data. Double-click the end nodes to display the color picker. Set the red endpoint to the RGB value [50 0 0] and set the blue endpoint to the RGB value [0 0 50].

The slice plane coloring now matches the example objectives.
Saving the Modified Colormap

You can save the modified colormap using the colormap function or the figure Colormap property.

After you have applied your changes, save the current figure colormap in a variable:

```matlab
mycmap = get(fig,'Colormap'); % fig is figure handle or use gcf
```

To use this colormap in another figure, set that figure's Colormap property:

```matlab
set(new_fig,'Colormap',mycmap)
```

To save your modified colormap in a MAT-file, use the save command to save the mycmap workspace variable:

```matlab
save('MyColormaps','mycmap')
```

To use your saved colormap in another MATLAB session, load the variable into the workspace and assign the colormap to the figure:

```matlab
load('MyColormaps','mycmap')
set(fig,'Colormap',mycmap)
```
See Also

`colormap`, `get`, `load`, `save`, `set`

Color Operations for related functions

See Colormaps for more information on using MATLAB colormaps.
**ColorSpec**

**Purpose**
Color specification

**Description**
ColorSpec is not a command; it refers to the three ways in which you specify color in MATLAB:

- RGB triple
- Short name
- Long name

The short names and long names are MATLAB strings that specify one of eight predefined colors. The RGB triple is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1]. The following table lists the predefined colors and their RGB equivalents.

<table>
<thead>
<tr>
<th>RGB Value</th>
<th>Short Name</th>
<th>Long Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1 1 0]</td>
<td>y</td>
<td>yellow</td>
</tr>
<tr>
<td>[1 0 1]</td>
<td>m</td>
<td>magenta</td>
</tr>
<tr>
<td>[0 1 1]</td>
<td>c</td>
<td>cyan</td>
</tr>
<tr>
<td>[1 0 0]</td>
<td>r</td>
<td>red</td>
</tr>
<tr>
<td>[0 1 0]</td>
<td>g</td>
<td>green</td>
</tr>
<tr>
<td>[0 0 1]</td>
<td>b</td>
<td>blue</td>
</tr>
<tr>
<td>[1 1 1]</td>
<td>w</td>
<td>white</td>
</tr>
<tr>
<td>[0 0 0]</td>
<td>k</td>
<td>black</td>
</tr>
</tbody>
</table>

**Remarks**
The eight predefined colors and any colors you specify as RGB values are not part of a figure's colormap, nor are they affected by changes to the figure's colormap. They are referred to as *fixed* colors, as opposed to *colormap* colors.

**Examples**
To change the background color of a figure to green, specify the color with a short name, a long name, or an RGB triple. These statements generate equivalent results:

- `whitebg('g')`
You can use `ColorSpec` anywhere you need to define a color. For example, this statement changes the figure background color to pink:

```matlab
set(gcf, 'Color', [1,0.4,0.6])
```

**See Also**

`bar`, `bar3`, `colordef`, `colormap`, `fill`, `fill3`, `whitebg`

“Color Operations” for related functions
Purpose
Sparse column permutation based on nonzero count

Syntax
j = colperm(S)

Description
j = colperm(S) generates a permutation vector j such that the columns of S(:,j) are ordered according to increasing count of nonzero entries. This is sometimes useful as a preordering for LU factorization; in this case use lu(S(:,j)).

If S is symmetric, then j = colperm(S) generates a permutation j so that both the rows and columns of S(j,j) are ordered according to increasing count of nonzero entries. If S is positive definite, this is sometimes useful as a preordering for Cholesky factorization; in this case use chol(S(j,j)).

Algorithm
The algorithm involves a sort on the counts of nonzeros in each column.

Examples
The n-by-n arrowhead matrix
A = [ones(1,n); ones(n-1,1) speye(n-1,n-1)]
has a full first row and column. Its LU factorization, lu(A), is almost completely full. The statement
j = colperm(A)
returns j = [2:n 1]. So A(j,j) sends the full row and column to the bottom and the rear, and lu(A(j,j)) has the same nonzero structure as A itself.

On the other hand, the Bucky ball example,
B = bucky
has exactly three nonzero elements in each row and column, so
j = colperm(B) is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.

See Also
chol, colamd, colmmd, lu, spparms, symamd, symmmd, symrcm
Purpose
Two-dimensional comet plot

Syntax
comet(y)
comet(x,y)
comet(x,y,p)
comet(axes_handle,...)

Description
A comet graph is an animated graph in which a circle (the comet head) traces the data points on the screen. The comet body is a trailing segment that follows the head. The tail is a solid line that traces the entire function.

comet(y) displays a comet graph of the vector y.

comet(x,y) displays a comet graph of vector y versus vector x.

comet(x,y,p) specifies a comet body of length p*length(y). p defaults to 0.1.

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

Remarks
Note that the trace left by comet is created by using an EraseMode of none, which means you cannot print the graph (you get only the comet head) and it disappears if you cause a redraw (e.g., by resizing the window).

Examples
Create a simple comet graph:

t = 0:.01:2*pi;
x = cos(2*t).*(cos(t).^2);
y = sin(2*t).*(sin(t).^2);
comet(x,y);

See Also
comet3
“Direction and Velocity Plots” for related functions
Purpose
Three-dimensional comet plot

Syntax
comet3(z)
comet3(x,y,z)
comet3(x,y,z,p)
comet3(axes_handle,...)

Description
A comet plot is an animated graph in which a circle (the comet head) traces the data points on the screen. The comet body is a trailing segment that follows the head. The tail is a solid line that traces the entire function.

comet3(z) displays a three-dimensional comet graph of the vector z.

comet3(x,y,z) displays a comet graph of the curve through the points [x(i),y(i),z(i)].

comet3(x,y,z,p) specifies a comet body of length p*length(y).

comet3(axes_handle,...) plots into the axes with handle axes_handle instead of the current axes (gca).

Remarks
Note that the trace left by comet3 is created by using an EraseMode of none, which means you cannot print the graph (you get only the comet head) and it disappears if you cause a redraw (e.g., by resizing the window).

Examples
Create a three-dimensional comet graph.

```matlab
t = -10*pi:pi/250:10*pi;
comet3((cos(2*t).^2).*sin(t),(sin(2*t).^2).*cos(t),t);
```

See Also
comet
“Direction and Velocity Plots” for related functions
commandhistory

**Purpose**
Open the Command History, or select it if already open

**Graphical Interface**
As an alternative to `commandhistory`, select Desktop -> **Command History** to open it, or Window -> **Command History** to select it.

**Syntax**
`commandhistory`

**Description**
`commandhistory` opens the MATLAB Command History when it is closed, and selects the Command History when it is open. The Command History presents a log of the statements most recently run in the Command Window.

Timestamp marks the start of each session. Select it to select all entries in the history for that session.

Click - to hide history for that session.
Click + to expand.

Select one or more lines and right-click to copy, evaluate, or create a shortcut or an M-file from the selection.

**See Also**
diary, startup -logfile option

“Recalling Previous Lines”

“Command History” in the MATLAB Desktop Tools documentation
**Purpose**
Open the Command Window, or select it if already open

**Graphical Interface**
As an alternative to `commandwindow`, select Desktop -> **Command Window** to open it, or Window -> **Command Window** to select it.

**Syntax**
`commandwindow`

**Description**
`commandwindow` opens the MATLAB Command Window when it is closed, and selects the Command Window when it is open.

**Remarks**
To determine the number of columns and rows that will display in the Command Window, given its current size, use

```
get(0,'CommandWindowSize')
```

The number of columns is based on the width of the Command Window. With the matrix display width preference set to 80 columns, the number of columns is always 80.

**See Also**
MATLAB Desktop Tools and Development Environment documentation

“Opening and Arranging Tools”

“Running Functions—Command Window and History”

“Preferences for the Command Window”
Purpose
Companion matrix

Syntax
A = compan(u)

Description
A = compan(u) returns the corresponding companion matrix whose first row is
-u(2:n)/u(1), where u is a vector of polynomial coefficients. The eigenvalues
of compan(u) are the roots of the polynomial.

Examples
The polynomial \((x - 1)(x - 2)(x + 3) = x^3 - 7x + 6\) has a companion matrix
given by

\[
\begin{align*}
u &= [1 \ 0 \ -7 \ 6] \\
A &= \text{compan}(u) \\
A &= \\
&= \begin{bmatrix}
0 & 7 & -6 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\end{align*}
\]

The eigenvalues are the polynomial roots:

\[
eig(\text{compan}(u))
\]

\[
\begin{align*}
\text{ans} &= \\
&= \begin{bmatrix}
-3.0000 \\
2.0000 \\
1.0000
\end{bmatrix}
\end{align*}
\]

This is also roots(u).

See Also
eig, poly, polyval, roots
Purpose

Plot arrows emanating from the origin

Syntax

compass(U,V)
compass(Z)
compass(...,LineSpec)
compass(axes_handle,...)
h = compass(...)

Description

A compass graph displays the vectors with components (U,V) as arrows emanating from the origin. U, V, and Z are in Cartesian coordinates and plotted on a circular grid.

compass(U,V) displays a compass graph having n arrows, where n is the number of elements in U or V. The location of the base of each arrow is the origin. The location of the tip of each arrow is a point relative to the base and determined by [U(i),V(i)].

compass(Z) displays a compass graph having n arrows, where n is the number of elements in Z. The location of the base of each arrow is the origin. The location of the tip of each arrow is relative to the base as determined by the real and imaginary components of Z. This syntax is equivalent to compass(real(Z),imag(Z)).

compass(...,LineSpec) draws a compass graph using the line type, marker symbol, and color specified by LineSpec.

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

h = compass(...) returns handles to line objects.

Examples

Draw a compass graph of the eigenvalues of a matrix.

Z = eig(randn(20,20));
compass(Z)
See Also  feather, LineSpec, quiver, rose

“Direction and Velocity Plots” for related functions
Compass Plots for another example

complex

Purpose  Construct complex data from real and imaginary components

Syntax  
\[
c = \text{complex}(a,b)
\]
\[
c = \text{complex}(a)
\]

Description  
\[
c = \text{complex}(a,b)
\]
creates a complex output, c, from the two real inputs.
\[
c = a + bi
\]

The output is the same size as the inputs, which must be scalars or equally sized vectors, matrices, or multi-dimensional arrays.
**Note** If b is all zeros, c is complex and the value of all its imaginary components is 0. In contrast, the result of the addition a+0i returns a strictly real result.

The following describes when a and b can have different data types, and the resulting data type of the output c:

- If either of a or b has type single, c has type single.
- If either of a or b has an integer data type, the other must have the same integer data type or type scalar double, and c has the same integer data type.

```
c = complex(a) for real a returns the complex result c with real part a and 0 as the value of all imaginary components. Even though the value of all imaginary components is 0, c is complex and isreal(c) returns false.
```

The complex function provides a useful substitute for expressions such as

```
a + i*b or a + j*b
```

in cases when the names “i” and “j” may be used for other variables (and do not equal $\sqrt{-1}$), when a and b are not single or double, or when b is all zero.

**Example**

Create complex uint8 vector from two real uint8 vectors.

```
a = uint8([1;2;3;4])
b = uint8([2;2;7;7])
c = complex(a,b)
c =
1.0000 + 2.0000i
2.0000 + 2.0000i
3.0000 + 7.0000i
4.0000 + 7.0000i
```

**See Also**

abs, angle, conj, i, imag, isreal, j, real
Purpose
Identify information about computer on which MATLAB is running

Syntax
str = computer
[str,maxsize] = computer
[str,maxsize,endian] = computer

Description
str = computer returns the string str with the computer type on which MATLAB is running.

[str,maxsize] = computer returns the integer maxsize, which contains the maximum number of elements allowed in an array with this version of MATLAB.

[str,maxsize,endian] = computer also returns either 'L' for little endian byte ordering or 'B' for big endian byte ordering.

The list of supported computers changes as new computers are added and others become obsolete. A typical list follows.

<table>
<thead>
<tr>
<th>str</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLNX86</td>
<td>Linux on PC</td>
</tr>
<tr>
<td>GLNXI64</td>
<td>Linux on Intel Itanium2</td>
</tr>
<tr>
<td>HPUX</td>
<td>HP PA-RISC (HP-UX 11.00)</td>
</tr>
<tr>
<td>MAC</td>
<td>Macintosh OS X</td>
</tr>
<tr>
<td>PCWIN</td>
<td>Microsoft Windows</td>
</tr>
<tr>
<td>SOL2</td>
<td>Sun Solaris 2 SPARC workstation</td>
</tr>
</tbody>
</table>

See Also
ispc, isunix
Purpose

Condition number with respect to inversion

Syntax

c = cond(X)
c = cond(X,p)

Description

The condition number of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and the linear equation solution. Values of \( \text{cond}(X) \) and \( \text{cond}(X,p) \) near 1 indicate a well-conditioned matrix.

\[
c = \text{cond}(X) \text{ returns the 2-norm condition number, the ratio of the largest singular value of } X \text{ to the smallest.}
\]

\[
c = \text{cond}(X,p) \text{ returns the matrix condition number in } p\text{-norm:}
\]

\[
\text{norm}(X,p) \times \text{norm}(\text{inv}(X),p)
\]

<table>
<thead>
<tr>
<th>If ( p ) is...</th>
<th>Then ( \text{cond}(X,p) ) returns the...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-norm condition number</td>
</tr>
<tr>
<td>2</td>
<td>2-norm condition number</td>
</tr>
<tr>
<td>'fro'</td>
<td>Frobenius norm condition number</td>
</tr>
<tr>
<td>inf</td>
<td>Infinity norm condition number</td>
</tr>
</tbody>
</table>

Algorithm

The algorithm for \( \text{cond} \) (when \( p = 2 \)) uses the singular value decomposition, \( \text{svd} \).

See Also

condeig, condest, norm, normest, rank, rcond, svd

References

Purpose
Condition number with respect to eigenvalues

Syntax
\[
c = \text{condeig}(A)
\]
\[
[V,D,s] = \text{condeig}(A)
\]

Description
\[c = \text{condeig}(A)\] returns a vector of condition numbers for the eigenvalues of \(A\). These condition numbers are the reciprocals of the cosines of the angles between the left and right eigenvectors.

\[[V,D,s] = \text{condeig}(A)\] is equivalent to
\[
[V,D] = \text{eig}(A);
\]
\[
s = \text{condeig}(A);
\]

Large condition numbers imply that \(A\) is near a matrix with multiple eigenvalues.

See Also
balance, cond, eig
### condest

**Purpose**

1-norm condition number estimate

**Syntax**

- `c = condest(A)`
- `[c,v] = condest(A)`

**Description**

- `c = condest(A)` computes a lower bound C for the 1-norm condition number of a square matrix A.

- `c = condest(A,t)` changes t, a positive integer parameter equal to the number of columns in an underlying iteration matrix. Increasing the number of columns usually gives a better condition estimate but increases the cost. The default is `t = 2`, which almost always gives an estimate correct to within a factor 2.

- `[c,v] = condest(A)` also computes a vector v which is an approximate null vector if c is large. v satisfies `norm(A*v,1) = norm(A,1)*norm(v,1)/c`.

**Note**

`condest` invokes `rand`. If repeatable results are required then invoke `rand('state', j)`, for some j, before calling this function.

This function is particularly useful for sparse matrices.

`condest` uses the block 1-norm power method of Higham and Tisseur [1].

**See Also**

- `cond`, `norm`, `normest`

**Reference**

**Purpose**
Plot velocity vectors as cones in a 3-D vector field

**Syntax**
```matlab
coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)
coneplot(U,V,W,Cx,Cy,Cz)
coneplot(...,s)
coneplot(...,color)
coneplot(...,'quiver')
coneplot(...,'method')
coneplot(X,Y,Z,U,V,W,'nointerp')
h = coneplot(axes_handle,...)
```

**Description**
`coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)` plots velocity vectors as cones pointing in the direction of the velocity vector and having a length proportional to the magnitude of the velocity vector.

- `X, Y, Z` define the coordinates for the vector field.
- `U, V, W` define the vector field. These arrays must be the same size, monotonic, and 3-D plaid (such as the data produced by `meshgrid`).
- `Cx, Cy, Cz` define the location of the cones in the vector field. The section Starting Points for Stream Plots in Visualization Techniques provides more information on defining starting points.

`coneplot(U,V,W,Cx,Cy,Cz)` (omitting the `X, Y,` and `Z` arguments) assumes

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p) where [m,n,p]= size(U).
```

`coneplot(...,s)` MATLAB automatically scales the cones to fit the graph and then stretches them by the scale factor `s`. If you do not specify a value for `s`, MATLAB uses a value of 1. Use `s = 0` to plot the cones without automatic scaling.

`coneplot(...,color)` interpolates the array `color` onto the vector field and then colors the cones according to the interpolated values. The size of the `color` array must be the same size as the `U, V, W` arrays. This option works only with cones (i.e., not with the `quiver` option).

`coneplot(...,'quiver')` draws arrows instead of cones (see `quiver3` for an illustration of a quiver plot).
coneplot(..., 'method') specifies the interpolation method to use. *method* can be *linear*, *cubic*, or *nearest*. *linear* is the default (see interp3 for a discussion of these interpolation methods).

coneplot(X,Y,Z,U,V,W, 'nointerp') does not interpolate the positions of the cones into the volume. The cones are drawn at positions defined by X, Y, Z and are oriented according to U, V, W. Arrays X, Y, Z, U, V, W must all be the same size.

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

h = coneplot(...) returns the handle to the patch object used to draw the cones. You can use the set command to change the properties of the cones.

Remarks

coneplot automatically scales the cones to fit the graph, while keeping them in proportion to the respective velocity vectors.

It is usually best to set the data aspect ratio of the axes before calling coneplot. You can set the ratio using the daspect command,

daspect([1,1,1])

Examples

This example plots the velocity vector cones for vector volume data representing the motion of air through a rectangular region of space. The final graph employs a number of enhancements to visualize the data more effectively. These include

- Cone plots indicate the magnitude and direction of the wind velocity.
- Slice planes placed at the limits of the data range provide a visual context for the cone plots within the volume.
- Directional lighting provides visual cues to the orientation of the cones.
- View adjustments compose the scene to best reveal the information content of the data by selecting the view point, projection type, and magnification.

1. Load and Inspect Data

The winds data set contains six 3-D arrays: u, v, and w specify the vector components at each of the coordinates specified in x, y, and z. The coordinates define a lattice grid structure where the data is sampled within the volume.
It is useful to establish the range of the data to place the slice planes and to specify where you want the cone plots (min, max).

```matlab
load wind
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
ymax = max(y(:));
zmin = min(z(:));
```

2. Create the Cone Plot

- Decide where in data space you want to plot cones. This example selects the full range of x and y in eight steps and the range 3 to 15 in four steps in z (linspace, meshgrid).
- Use daspect to set the data aspect ratio of the axes before calling `coneplot` so MATLAB can determine the proper size of the cones.
- Draw the cones, setting the scale factor to 5 to make the cones larger than the default size.
- Set the coloring of each cone (FaceColor, EdgeColor).

```matlab
daspect([2,2,1])
xrange = linspace(xmin,xmax,8);
yrange = linspace(ymin,ymax,8);
zrange = 3:4:15;
[x y z] = meshgrid(xrange,yrange,zrange);
[cx cy cz] = meshgrid(xrange,yrange,zrange);
hcones = coneplot(x,y,z,u,v,w,cx, cy, cz, 5);
set(hcones, 'FaceColor', 'red', 'EdgeColor', 'none')
```
3. Add the Slice Planes

- Calculate the magnitude of the vector field (which represents wind speed) to generate scalar data for the slice command.
- Create slice planes along the x-axis at $x_{\text{min}}$ and $x_{\text{max}}$, along the y-axis at $y_{\text{max}}$, and along the z-axis at $z_{\text{min}}$.
- Specify interpolated face color so the slice coloring indicates wind speed and do not draw edges (hold, slice, FaceColor, EdgeColor).

```matlab
hold on
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hsurfaces = slice(x,y,z,wind_speed,[xmin,xmax],ymax,zmin);
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
hold off
```

4. Define the View

- Use the axis command to set the axis limits equal to the range of the data.
- Orient the view to azimuth = 30 and elevation = 40 (rotate3d is a useful command for selecting the best view).
- Select perspective projection to provide a more realistic looking volume (camproj).
- Zoom in on the scene a little to make the plot as large as possible (camzoom).

```matlab
axis tight; view(30,40); axis off
camproj perspective; camzoom(1.5)
```

5. Add Lighting to the Scene

The light source affects both the slice planes (surfaces) and the cone plots (patches). However, you can set the lighting characteristics of each independently.

- Add a light source to the right of the camera and use Phong lighting to give the cones and slice planes a smooth, three-dimensional appearance (camlight, lighting).
- Increase the value of the AmbientStrength property for each slice plane to improve the visibility of the dark blue colors. (Note that you can also specify a different colormap to change the coloring of the slice planes.)
Increase the value of the DiffuseStrength property of the cones to brighten particularly those cones not showing specular reflections.

```
camlight right; lighting phong
set(hsurfaces,'AmbientStrength',.6)
set(hcones,'DiffuseStrength',.8)
```

See Also

isosurface, patch, reducevolume, smooth3, streamline, stream2, stream3, subvolume

“Volume Visualization” for related functions
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Complex conjugate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>( Z' = \text{conj}(Z) )</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>( Z' = \text{conj}(Z) ) returns the complex conjugate of the elements of ( Z ).</td>
</tr>
</tbody>
</table>
| **Algorithm** | If \( Z \) is a complex array:
  \[
  \text{conj}(Z) = \text{real}(Z) - i \cdot \text{imag}(Z)
  \] |
| **See Also** | \( i, j, \text{imag}, \text{real} \) |
**Purpose**
Pass control to the next iteration of `for` or `while` loop

**Syntax**
`continue`

**Description**
`continue` passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop.

In nested loops, `continue` passes control to the next iteration of the `for` or `while` loop enclosing it.

**Examples**
The example below shows a `continue` loop that counts the lines of code in the file `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered.

```matlab
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line,'%',1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines',count));
```

**See Also**
`for`, `while`, `end`, `break`, `return`
**Purpose**

Contour graph of a matrix

**Syntax**

- `contour(Z)`
- `contour(Z,n)`
- `contour(Z,v)`
- `contour(X,Y,Z)`
- `contour(X,Y,Z,n)`
- `contour(X,Y,Z,v)`
- `contour(...,LineSpec)`
- `[C,h] = contour(...)`
- `[C,h] = contour('v6',...)`

**Description**

A contour graph displays isolines of matrix Z. Label the contour lines using `clabel`.

`contour(Z)` draws a contour plot of matrix Z, where Z is interpreted as heights with respect to the x-y plane. Z must be at least a 2-by-2 matrix. The number of contour levels and the values of the contour levels are chosen automatically based on the minimum and maximum values of Z. The ranges of the x- and y-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contour(Z,n)` draws a contour plot of matrix Z with n contour levels.

`contour(Z,v)` draws a contour plot of matrix Z with contour lines at the data values specified in vector v. The number of contour levels is equal to `length(v)`. To draw a single contour of level i, use `contour(Z,[i i])`.

`contour(X,Y,Z), contour(X,Y,Z,n), and contour(X,Y,Z,v)` draw contour plots of Z. X and Y specify the x- and y-axis limits. When X and Y are matrices, they must be the same size as Z, in which case they specify a surface, as defined by the `surf` function.

If X or Y is irregularly spaced, `contour` calculates contours using a regularly spaced contour grid, then transforms the data to X or Y.

`contour(...,LineSpec)` draws the contours using the line type and color specified by `LineSpec`. `contour` ignores marker symbols.
\([C,h] = \text{contour}(\ldots)\) returns the contour matrix \(C\) (see \text{contourc}) and a handle to a contourgroup object. \text{clabel} uses the contour matrix \(C\) to create the labels. (See descriptions of contourgroup object properties.)

**Backward Compatible Version**

\([C,h] = \text{contour}(\text{'v6'},\ldots)\) returns the contour matrix \(C\) (see \text{contourc}) and a vector of handles to graphics objects. \text{clabel} uses the contour matrix \(C\) to create the labels. \text{contour} creates patch graphics objects unless you specify a LineSpec, in which case \text{contour} creates line graphics objects.

See Plot Objects and Backward Compatibility for more information.

**Remarks**

If you do not specify the LineSpec argument, the figure colormap (\text{colormap}) and the color limits (\text{caxis}) control the color of the contour lines. In this case the contour function creates patch objects to implement the contour plot.

When you specify the LineSpec argument, the \text{contour} function creates line object to implement the contour plot. In this case, contour lines are not mapped to colors in the figure colormap, but are colored using the colors defined in the axes ColorOrder property.

Use contourgroup object properties to control the contour plot appearance.

The following diagram illustrates the parent-child relationship in contour plots.
Examples

Contour Plot of a Function
To view a contour plot of the function

$$z = xe^{(-x^2 - y^2)}$$

over the range $-2 \leq x \leq 2$, $-2 \leq y \leq 3$, create matrix $Z$ using the statements

$$[X,Y] = \text{meshgrid}(-2:.2:2,-2:.2:3);$$
$$Z = X.*\exp(-X.^2-Y.^2);$$

Then, generate a contour plot of $Z$.

- Display contour labels by setting the ShowText property to on.
- Label every other contour line by setting the TextStep property to twice the contour interval (i.e., two times the LevelStep property).
- Use a smoothly varying colormap.

$$[C,h] = \text{contour}(X,Y,Z);$$
$$\text{set}(h, 'ShowText', 'on', 'TextStep', \text{get}(h, 'LevelStep')*2)$$
$$\text{colormap cool}$$
Smoothing Contour Data

You can use interp2 to create smoother contours. Also set the contour label textBackgroundColor to a light yellow and the EdgeColor to light gray.

```matlab
Z = peaks;
[C,h] = contour(interp2(Z,4));
text_handle = clabel(C,h);
set(text_handle,'BackgroundColor',[1 1 .6],...
    'Edgecolor',[.7 .7 .7])
```
Setting the Axis Limits on Contour Plots

Suppose, for example, your data represents a region that is 1000 meters in the \(x\) dimension and 3000 meters in the \(y\) dimension. You could use the following statements to set the axis limits correctly:

```matlab
Z = rand(24,36); % assume data is a 24-by-36 matrix
X = linspace(0,1000,size(Z,2));
Y = linspace(0,3000,size(Z,1));
[c,h] = contour(X,Y,Z);
axis equal tight % set the axes aspect ratio
```

See Also

contour3, contourc, contourf, contourslice

See “Contourgroup Properties” for property descriptions
**Purpose**

Three-dimensional contour plot

**Syntax**

- `contour3(Z)`
- `contour3(Z,n)`
- `contour3(Z,v)`
- `contour3(X,Y,Z)`
- `contour3(X,Y,Z,n)`
- `contour3(X,Y,Z,v)`
- `contour3(axes_handle,...)`
- `contour3(...,LineSpec)`
- `[C,h] = contour3(...)`

**Description**

`contour3` creates a three-dimensional contour plot of a surface defined on a rectangular grid.

`contour3(Z)` draws a contour plot of matrix `Z` in a three-dimensional view. `Z` is interpreted as heights with respect to the `x-y` plane. `Z` must be at least a 2-by-2 matrix. The number of contour levels and the values of contour levels are chosen automatically. The ranges of the `x`- and `y`-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z).

`contour3(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels in a three-dimensional view.

`contour3(Z,v)` draws a contour plot of matrix `Z` with contour lines at the values specified in vector `v`. The number of contour levels is equal to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i]).`

`contour3(X,Y,Z)`, `contour3(X,Y,Z,n)`, and `contour3(X,Y,Z,v)` use `X` and `Y` to define the `x`- and `y`-axis limits. If `X` is a matrix, `X(1,:)` defines the `x`-axis. If `Y` is a matrix, `Y(:,1)` defines the `y`-axis. When `X` and `Y` are matrices, they must be the same size as `Z`, in which case they specify a surface as `surf` does.

`contour3(...,LineSpec)` draws the contours using the line type and color specified by `LineSpec`.

`contour3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).
\texttt{contour3} \par

\texttt{[C,h] = contour3(...) } returns the contour matrix \texttt{C} as described in the function \texttt{contourc} and a column vector containing handles to graphics objects. \texttt{contour3} creates patch graphics objects unless you specify \texttt{LineSpec}, in which case \texttt{contour3} creates line graphics objects. \par

\textbf{Remarks} \par

If you do not specify \texttt{LineSpec}, \texttt{colormap} and \texttt{caxis} control the color. \par

If \texttt{X} or \texttt{Y} is irregularly spaced, \texttt{contour3} calculates contours using a regularly spaced contour grid, then transforms the data to \texttt{X} or \texttt{Y}. \par

\textbf{Examples} \par

Plot the three-dimensional contour of a function and superimpose a surface plot to enhance visualization of the function. \par

\begin{verbatim}
[X,Y] = meshgrid([-2:.25:2]);
Z = X.*exp(-X.^2-Y.^2);
contour3(X,Y,Z,30)
surface(X,Y,Z,'EdgeColor',[.8 .8 .8],'FaceColor','none')
grid off
view(-15,25)
colormap cool
\end{verbatim}
See Also

contour, contourc, meshc, meshgrid, surfc

“Contour Plots” category for related functions

Contour Plots section for more examples
contourc

Purpose
Low-level contour plot computation

Syntax
C = contourc(Z)
C = contourc(Z,n)
C = contourc(Z,v)
C = contourc(x,y,Z)
C = contourc(x,y,Z,n)
C = contourc(x,y,Z,v)

Description
contourc calculates the contour matrix C used by contour, contour3, and contourf. The values in Z determine the heights of the contour lines with respect to a plane. The contour calculations use a regularly spaced grid determined by the dimensions of Z.

C = contourc(Z) computes the contour matrix from data in matrix Z, where Z must be at least a 2-by-2 matrix. The contours are isolines in the units of Z. The number of contour lines and the corresponding values of the contour lines are chosen automatically.

C = contourc(Z,n) computes contours of matrix Z with n contour levels.

C = contourc(Z,v) computes contours of matrix Z with contour lines at the values specified in vector v. The length of v determines the number of contour levels. To compute a single contour of level i, use contourc(Z,[i i]).

C = contourc(x,y,Z), C = contourc(x,y,Z,n), and C = contourc(x,y,Z,v) compute contours of Z using vectors x and y to determine the x- and y-axis limits. x and y must be monotonically increasing.

Remarks
C is a two-row matrix specifying all the contour lines. Each contour line defined in matrix C begins with a column that contains the value of the contour (specified by v and used by clabel), and the number of (x,y) vertices in the contour line. The remaining columns contain the data for the (x,y) pairs.

C = [value1 xdata(1) xdata(2)...value2 xdata(1) xdata(2)...;
     dim1 ydata(1) ydata(2)...dim2 ydata(1) ydata(2)...
]

Specifying irregularly spaced x and y vectors is not the same as contouring irregularly spaced data. If x or y is irregularly spaced, contourc calculates
contours using a regularly spaced contour grid, then transforms the data to x or y.

See Also

clabel, contour, contour3, contourf

“Contour Plots” for related functions

The Contouring Algorithm for more information
contourf

Purpose
Filled two-dimensional contour plot

Syntax
contourf(Z)
contourf(Z,n)
contourf(Z,v)
contourf(X,Y,Z)
contourf(X,Y,Z,n)
contourf(X,Y,Z,v)
contourf(axes_handle,...)
[C,h,CF] = contourf(...)

Description
A filled contour plot displays isolines calculated from matrix Z and fills the areas between the isolines using constant colors. The color of the filled areas depends on the current figure's colormap.

contourf(Z) draws a contour plot of matrix Z, where Z is interpreted as heights with respect to a plane. Z must be at least a 2-by-2 matrix. The number of contour lines and the values of the contour lines are chosen automatically.

contourf(Z,n) draws a contour plot of matrix Z with n contour levels.

contourf(Z,v) draws a contour plot of matrix Z with contour levels at the values specified in vector v.

contourf(X,Y,Z), contourf(X,Y,Z,n), and contourf(X,Y,Z,v) produce contour plots of Z using X and Y to determine the x- and y-axis limits. When X and Y are matrices, they must be the same size as Z, in which case they specify a surface as surf does.

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

[C,h,CF] = contourf(...) returns the contour matrix C as calculated by the function contourc and used by clabel, a vector of handles h to patch graphics objects, and a contour matrix CF for the filled areas.

Remarks
If X or Y is irregularly spaced, contourf calculates contours using a regularly spaced contour grid, then transforms the data to X or Y.
**Examples**

Create a filled contour plot of the peaks function.

```matlab
[C,h] = contourf(peaks(20),10);
colormap autumn
```

See Also: clabel, contour, contour3, contourc, quiver

“Contour Plots” for related functions
Contourgroup Properties

Modifying Properties
You can set and query graphics object properties using the set and get commands or the Property Editor (propertyeditor).

Note that you cannot define default properties for contourgroup objects.

See Plot Objects for more information on contourgroup objects.

Contourgroup 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.
Contourgroup 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 contourgroup object, but not over another graphics object. See the HitTestArea property for information about selecting contourgroup objects.

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

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

Clipping  {on} | off

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

ContourMatrix  2-by-n matrix

*A two-row matrix specifying all the contour lines.* Each contour line defined in the ContourMatrix begins with a column that contains the value of the contour (specified by the LevelList property and is used by clabel), and the number of \((x,y)\) vertices in the contour line. The remaining columns contain the data for the \((x,y)\) pairs:

```matlab
    C = [value1 xdata(1) xdata(2)...value2 xdata(1) xdata(2)...;
         dim1 ydata(1) ydata(2)...dim2 ydata(1) ydata(2)...]```
Contourgroup Properties

CreateFcn

string or function handle

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

```matlab
contour(Z, 'CreateFcn', @CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function.

MATLAB executes this routine after setting all other contourgroup properties. Setting this property on an existing contourgroup 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 contourgroup object is deleted (e.g., this might happen when you issue a `delete` command on the contourgroup 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.

DisplayName

string

Label used by plot legends. The legend and the plot browser uses this text for labels for any contourgroup objects appearing in these legends.

EraseMode

{normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase contour 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.
Contourgroup Properties

- **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 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 may mathematically combine layers of colors (e.g., performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

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

**Fill**  {off} | on

*Color spaces between contour lines.* By default, contour draws only the contour lines of the surface. If you set Fill to on, contour colors the regions in between the contour lines according to the Z-value of the region and changes the contour lines to black.

**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 contourgroup 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 whether the contourgroup 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 contour plot. If HitTest is off, clicking the contour selects the object below it (which is usually the axes containing it).

HitTestArea  on | {off}
Select contourgroup object on contour lines or area of extent. This property enables you to select contourgroup objects in two ways:

• Select by clicking contour lines (default).
• Select by clicking anywhere in the extent of the contour plot.

When HitTestArea is off, you must click the contour lines (excluding the baseline) to select the contourgroup object. When HitTestArea is on, you can select the contourgroup object by clicking anywhere within the extent of the contour plot (i.e., anywhere within a rectangle that encloses all the contour lines).

Interruptible  {on} | off
Callback routine interruption mode. The Interruptible property controls whether a contourgroup 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.
Contourgroup Properties

Setting **Interruptible** to on allows any graphics object’s callback to interrupt callback routines originating from a contour 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.

**LabelSpacing** distance in points (default = 144)

*Spacing between labels on each contour line.* When you display contour line labels using either the `ShowText` property or the `clabel` command, the labels are spaced 144 points (2 inches) apart on each line. You can specify the spacing by setting the `LabelSpacing` property to a value in points. If the length of an individual contour line is less than the specified value, MATLAB displays only one contour label on that line.

**LevelList** vector of `ZData`-values

*Values at which contour lines are drawn.* When the `LevelListMode` property is auto, the `contour` function automatically chooses contour values that span the range of values in `ZData` (the input argument `Z`). You can set this property to the values at which you want contour lines drawn.

To specify the contour interval (space between contour lines) use the `LevelStep` property.

**LevelListMode** {auto} | manual

*User-specified or autogenerated LevelList values.* By default, the `contour` function automatically generates the values at which contours are drawn. If you set this property to manual, `contour` does not change the values in `LevelList` as you change the values of `ZData`.

**LevelStep** scalar

*Spacing of contour lines.* The `contour` function draws contour lines at regular intervals determined by the value of `LevelStep`. When the `LevelStepMode` property is set to auto, `contour` determines the contour interval automatically based on the `ZData`.

**LevelStepMode** {auto} | manual

*User-specified or autogenerated LevelStep values.* By default, the `contour` function automatically determines a value for the `LevelStep` property. If you set this property to manual, `contour` does not change the value of `LevelStep` as you change the values of `ZData`. 
Contourgroup Properties

**LineColor**  
\{auto\} | ColorSpec | none

Color of the contour lines. This property determines how MATLAB colors the contour lines.

- **auto**— Each contour line is a single color determined by its contour value, the figure colormap, and the color axis (caxis).
- **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**— No contour lines are drawn.

**LineStyle**  
\{-\} | -- | : | -. | none

Line style. This property specifies the line style used for the contour 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 useLineStyle none when you want to place a marker at each point but do not want the points connected with a line.

**LineWidth**  
scalar

The width of the contour lines. Specify this value in points (1 point = \(\frac{1}{72}\) inch). The default LineWidth is 0.5 points.

**Parent**  
object handle

Parent of contourgroup object. This property contains the handle of the contourgroup object’s parent object. The parent of a contourgroup object is the axes, hggroup, or hgtransform object that contains it.
Contourgroup Properties

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 contourgroup 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 four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

**ShowText** on | {off}

Display labels on contour lines. When you set this property to on, MATLAB displays text labels on each contour line indicating the contour value. See also LevelList, clabel, and the example “Contour Plot of a Function”.

**Tag** string

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

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

```matlab
% t = contour('Tag','contour1')
```

When you want to access the contourgroup object, you can use findobj to find the contourgroup object’s handle. The following statement changes the MarkerFaceColor property of the object whose Tag is contour1.

```matlab
set(findobj('Tag','contour1'),'MarkerFaceColor','red')
```

**TextList** vector of contour values

*Contour values to label.* This property contains the contour values where text labels are placed. By default, these values are the same as those contained in
Contourgroup Properties

the LevelList property, which define where the contour lines are drawn. Note that there must be an equivalent contour line to display a text label.

For example, the following statements create and label a contour graph:

\[
[c,h]=\text{contour}(\text{peaks});
\]
\[
\text{clabel}(c,h)
\]

You can get the LevelList property to see the contour line values:

\[
\text{get}(h,'\text{LevelList}')
\]

Suppose you want to view the contour value 4.375 instead of the value of 4 that the contour function used. To do this, you need to set both the LevelList and TextList properties:

\[
\text{set}(h, '\text{LevelList}', [-6 -4 -2 0 2 4.375 6 8], ...
\]
\[
'\text{TextList}', [-6 -4 -2 0 2 4.375 6 8])
\]

See the example “Contour Plot of a Function” for additional information.

**TextListMode**

User-specified or auto TextList values. When this property is set to auto, MATLAB sets the TextList property equal to the values of the LevelList property (i.e., a text label for each contour line). When this property is set to manual, MATLAB does not set the values of the TextList property. Note that specifying values for the TextList property causes the TextListMode property to be set to manual.

**TextStep**

Determines which contour line have numeric labels. The contour function labels contour lines at regular intervals which are determined by the value of the TextStep property. When the TextStepMode property is set to auto, contour labels every contour line when the ShowText property is on. See “Contour Plot of a Function” for an example that uses the TextStep property.

**TextStepMode**

User-specified or autogenerated TextStep values. By default, the contour function automatically determines a value for the TextStep property. If you set this property to manual, contour does not change the value of TextStep as you change the values of ZData.
### Contourgroup Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>string (read only)</td>
</tr>
<tr>
<td><strong>UIContextMenu</strong></td>
<td>handle of a uicontextmenu object</td>
</tr>
<tr>
<td><strong>UserData</strong></td>
<td>array</td>
</tr>
<tr>
<td><strong>Visible</strong></td>
<td>{on}</td>
</tr>
<tr>
<td><strong>XData</strong></td>
<td>vector or matrix</td>
</tr>
</tbody>
</table>

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

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

**UIContextMenu**

Associate a context menu with the contourgroup object. Assign this property the handle of a uicontextmenu object created in the contourgroup 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 contourgroup object.

**UserData**

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

**Visible**

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

**XData**

X-axis limits. This property determines the x-axis limits used in the contour plot. If you do not specify an X argument, the `contour` function calculates x-axis limits based on the size of the input argument Z.

XData can be either a matrix equal in size to ZData or a vector equal in length to the number of rows in ZData.

Use XData to define meaningful coordinates for the underlying surface whose topography is being mapped. See “Setting the Axis Limits on Contour Plots” for more information.
Contourgroup Properties

XDataMode  
\{auto\} | manual

Use automatic or user-specified x-axis values. In auto mode (the default) the contour function automatically determines the x-axis limits. If you set this property to manual, specify a value for XData, or specify an X argument, then contour sets this property to manual and does not change the axis limits.

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  
scalar, vector, or matrix

Y-axis limits. This property determines the y-axis limits used in the contour plot. If you do not specify a Y argument, the contour function calculates y-axis limits based on the size of the input argument Z.

YData can be either a matrix equal in size to ZData or a vector equal in length to the number of columns in ZData.

Use YData to define meaningful coordinates for the underlying surface whose topography is being mapped. See “Setting the Axis Limits on Contour Plots” for more information.

YDataMode  
\{auto\} | manual

Use automatic or user-specified y-axis values. In auto mode (the default) the contour function automatically determines the y-axis limits. If you set this
Contourgroup Properties

property to manual, specify a value for YData, or specify a Y argument, then
contour sets this property to manual and does not change the axis limits.

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

Contour data. This property contains the data from which the contour lines are
generated (specified as the input argument Z). ZData must be at least a 2-by-2
matrix. The number of contour levels and the values of the contour levels are
chosen automatically based on the minimum and maximum values of ZData.
The limits of the x- and y-axis are [1:n] and [1:m], where [m,n] =
size(ZData).

ZDataSource

string (MATLAB variable)

Link ZData to MATLAB variable. Set this property to a MATLAB variable that
is evaluated in the base workspace to generate the ZData.

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

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

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

**Purpose**

Draw contours in volume slice planes

**Syntax**

`contourslice(X,Y,Z,V,Sx,Sy,Sz)`  
`contourslice(X,Y,Z,V,Xi,Yi,Zi)`  
`contourslice(V,Sx,Sy,Sz), contourslice(V,Xi,Yi,Zi)`  
`contourslice(...,n)`  
`contourslice(...,cvals)`  
`contourslice(...,[cv cv])`  
`contourslice(...,'method')`  
`contourslice(axes_handle,...)`  
`h = contourslice(...)`

**Description**

`contourslice(X,Y,Z,V,Sx,Sy,Sz)` draws contours in the x-, y-, and z-axis aligned planes at the points in the vectors Sx, Sy, Sz. The arrays X, Y, and Z define the coordinates for the volume V and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). The color at each contour is determined by the volume V, which must be an m-by-n-by-p volume array.

`contourslice(X,Y,Z,V,Xi,Yi,Zi)` draws contours through the volume V along the surface defined by the 2-D arrays Xi, Yi, Zi. The surface should lie within the bounds of the volume.

`contourslice(V,Sx,Sy,Sz)` and `contourslice(V,Xi,Yi,Zi)` (omitting the X, Y, and Z arguments) assume `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(v).

`contourslice(...,n)` draws n contour lines per plane, overriding the automatic value.

`contourslice(...,cvals)` draws `length(cval)` contour lines per plane at the values specified in vector cvals.

`contourslice(...,[cv cv])` computes a single contour per plane at the level cv.

`contourslice(...,'method')` specifies the interpolation method to use. `method` can be `linear`, `cubic`, or `nearest`. `nearest` is the default except when the contours are being drawn along the surface defined by Xi, Yi, Zi, in which case `linear` is the default (see `interp3` for a discussion of these interpolation methods).
contourslice(axes_handle,...) plots into the axes with handle axes_handle instead of the current axes (gca).

h = contourslice(...) returns a vector of handles to patch objects that are used to implement the contour lines.

Examples

This example uses the flow data set to illustrate the use of contoured slice planes (type doc flow for more information on this data set). Notice that this example

- Specifies a vector of length = 9 for Sx, an empty vector for the Sy, and a scalar value (0) for Sz. This creates nine contour plots along the x direction in the y-z plane, and one in the x-y plane at z = 0.
- Uses linspace to define a ten-element vector of linearly spaced values from -8 to 2. This vector specifies that ten contour lines be drawn, one at each element of the vector.
- Defines the view and projection type (camva, camproj, campos).
- Sets figure (gcf) and axes (gca) characteristics.

```matlab
[x y z v] = flow;
h = contourslice(x,y,z,v,[1:9],[],[0],linspace(-8,2,10));
axis([0,10,-3,3,-3,3]); daspect([1,1,1])
camva(24); camproj perspective;
campos([-3,-15,5])
set(gcf,'Color',[.5,.5,.5],'Renderer','zbuffer')
set(gca,'Color','black','XColor','white', ...  
  'YColor','white','ZColor','white')
box on
```
This example draws contour slices along a spherical surface within the volume.

```matlab
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2); % Create volume data
[xi,yi,zi] = sphere; % Plane to contour
contourslice(x,y,z,v,xi,yi,zi)
view(3)
```

See Also

isosurface, slice, smooth3, subvolume, reducevolume

“Volume Visualization” for related functions
Purpose

Grayscale colormap for contrast enhancement

Syntax

\[
\text{cmap} = \text{contrast}(X) \\
\text{cmap} = \text{contrast}(X,m)
\]

Description

The `contrast` function enhances the contrast of an image. It creates a new gray colormap, `cmap`, that has an approximately equal intensity distribution. All three elements in each row are identical.

\[
\text{cmap} = \text{contrast}(X) \text{ returns a gray colormap that is the same length as the current colormap.}
\]

\[
\text{cmap} = \text{contrast}(X,m) \text{ returns an m-by-3 gray colormap.}
\]

Examples

Add contrast to the clown image defined by `X`.

\[
\begin{align*}
\text{load clown; } \\
\text{cmap} = \text{contrast}(X); \\
\text{image(X); } \\
\text{colormap(cmap);}
\end{align*}
\]

See Also

`brighten`, `colormap`, `image`

“Colormaps” for related functions
**conv**

**Purpose**
Convolution and polynomial multiplication

**Syntax**
\[ w = \text{conv}(u,v) \]

**Description**
\[ w = \text{conv}(u,v) \] convolves vectors \( u \) and \( v \). Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of \( u \) and \( v \).

**Definition**
Let \( m = \text{length}(u) \) and \( n = \text{length}(v) \). Then \( w \) is the vector of length \( m+n-1 \) whose \( k \)th element is

\[
w(k) = \sum_{j} u(j)v(k+1-j)
\]

The sum is over all the values of \( j \) which lead to legal subscripts for \( u(j) \) and \( v(k+1-j) \), specifically \( j = \max(1,k+1-n) : \min(k,m) \). When \( m = n \), this gives

\[
\begin{align*}
w(1) &= u(1)v(1) \\
w(2) &= u(1)v(2) + u(2)v(1) \\
w(3) &= u(1)v(3) + u(2)v(2) + u(3)v(1) \\
&\quad \vdots \\
w(n) &= u(1)v(n) + u(2)v(n-1) + \cdots + u(n)v(1) \\
&\quad \vdots \\
w(2n-1) &= u(n)v(n)
\end{align*}
\]

**Algorithm**
The convolution theorem says, roughly, that convolving two sequences is the same as multiplying their Fourier transforms. In order to make this precise, it is necessary to pad the two vectors with zeros and ignore roundoff error. Thus, if

\[
X = \text{fft}([x \text{ zeros}(1,\text{length}(y)-1)])
\]

and

\[
Y = \text{fft}([y \text{ zeros}(1,\text{length}(x)-1)])
\]

then \( \text{conv}(x,y) = \text{ifft}(X.*Y) \)

**See Also**
conv2, convn, deconv, filter
convmtx and xcorr in the Signal Processing Toolbox
**Purpose**
Two-dimensional convolution

**Syntax**

\[ C = \text{conv2}(A,B) \]
\[ C = \text{conv2}(hcol,hrow,A) \]
\[ C = \text{conv2}(\ldots,\text{shape}) \]

**Description**

\[ C = \text{conv2}(A,B) \] computes the two-dimensional convolution of matrices \( A \) and \( B \). If one of these matrices describes a two-dimensional finite impulse response (FIR) filter, the other matrix is filtered in two dimensions.

The size of \( C \) in each dimension is equal to the sum of the corresponding dimensions of the input matrices, minus one. That is, if the size of \( A \) is \([ma,na]\) and the size of \( B \) is \([mb,nb]\), then the size of \( C \) is \([ma+mb-1,na+nb-1]\).

\[ C = \text{conv2}(hcol,hrow,A) \] convolves \( A \) first with the vector \( hcol \) along the rows and then with the vector \( hrow \) along the columns. If \( hcol \) is a column vector and \( hrow \) is a row vector, this case is the same as \( C = \text{conv2}(hcol*hrow,A) \).

\[ C = \text{conv2}(\ldots,\text{shape}) \] returns a subsection of the two-dimensional convolution, as specified by the shape parameter:

- **full**  Returns the full two-dimensional convolution (default).
- **same**  Returns the central part of the convolution of the same size as \( A \).
- **valid** Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, \( C \) has size \([ma-mb+1,na-nb+1]\) when all(size(A) >= size(B)). Otherwise \( \text{conv2} \) returns [].

**Algorithm**

\( \text{conv2} \) uses a straightforward formal implementation of the two-dimensional convolution equation in spatial form. If \( a \) and \( b \) are functions of two discrete variables, \( n_1 \) and \( n_2 \), then the formula for the two-dimensional convolution of \( a \) and \( b \) is

\[
c(n_1,n_2) = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} a(k_1,k_2) b(n_1-k_1,n_2-k_2)
\]

In practice however, \( \text{conv2} \) computes the convolution for finite intervals.
Note that matrix indices in MATLAB always start at 1 rather than 0. Therefore, matrix elements \(A(1,1)\), \(B(1,1)\), and \(C(1,1)\) correspond to mathematical quantities \(a(0,0)\), \(b(0,0)\), and \(c(0,0)\).

**Examples**

**Example 1.** For the 'same' case, \texttt{conv2} returns the central part of the convolution. If there are an odd number of rows or columns, the "center" leaves one more at the beginning than the end.

This example first computes the convolution of \(A\) using the default ('full') shape, then computes the convolution using the 'same' shape. Note that the array returned using 'same' corresponds to the underlined elements of the array returned using the default shape.

\[
A = \text{rand}(3); \\
B = \text{rand}(4); \\
C = \text{conv2}(A,B) \quad \% \ C \ is \ 6\text{-by-6}
\]

\[
C = \\
0.1838 \ 0.2374 \ 0.9727 \ 1.2644 \ 0.7890 \ 0.3750 \\
0.6929 \ 1.2019 \ 1.5499 \ 2.1733 \ 1.3325 \ 0.3096 \\
0.5627 \ 1.5150 \ 2.3576 \ 3.1553 \ 2.5373 \ 1.0602 \\
0.9986 \ 2.3811 \ 3.4302 \ 3.5128 \ 2.4489 \ 0.8462 \\
0.3089 \ 1.1419 \ 1.8229 \ 2.1561 \ 1.6364 \ 0.6841 \\
0.3287 \ 0.9347 \ 1.6464 \ 1.7928 \ 1.2422 \ 0.5423
\]

\[
Cs = \text{conv2}(A,B,'same') \quad \% \ Cs \ is \ the \ same \ size \ as \ A: \ 3\text{-by-3}
\]

\[
Cs = \\
2.3576 \ 3.1553 \ 2.5373 \\
3.4302 \ 3.5128 \ 2.4489 \\
1.8229 \ 2.1561 \ 1.6364
\]

**Example 2.** In image processing, the Sobel edge finding operation is a two-dimensional convolution of an input array with the special matrix

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

These commands extract the horizontal edges from a raised pedestal.

\[
A = \text{zeros}(10); \\
A(3:7,3:7) = \text{ones}(5); \\
H = \text{conv2}(A,s); \\
\text{mesh}(H)
\]
Transposing the filter \( s \) extracts the vertical edges of \( A \).

\[
V = \text{conv2}(A, s');
\]

\[
\text{figure, mesh}(V)
\]
**conv2**

This figure combines both horizontal and vertical edges.

```matlab
figure
mesh(sqrt(H.^2 + V.^2))
```

**See Also**

conv, convn, filter2

xcorr2 in the Signal Processing Toolbox
**Purpose**
Convex hull

**Syntax**
- `K = convhull(x,y)`
- `K = convhull(x,y,options)`
- `[K,a] = convhull(...)`

**Description**
- `K = convhull(x,y)` returns indices into the `x` and `y` vectors of the points on the convex hull. `convhull` uses Qhull.
- `K = convhull(x,y,options)` specifies a cell array of strings `options` to be used in Qhull via `convhulln`. The default option is `{'Qt'}`.
- If `options` is `[ ]`, the default options are used. If `options` is `''`, no options will be used, not even the default. For more information on Qhull and its options, see [http://www.qhull.org](http://www.qhull.org).
- `[K,a] = convhull(...)` also returns the area of the convex hull.

**Visualization**
Use `plot` to plot the output of `convhull`.

**Examples**
```matlab
xx = -1:.05:1; yy = abs(sqrt(xx));
[x,y] = pol2cart(xx,yy);
k = convhull(x,y);
plot(x(k),y(k),'r-',x,y,'b+')
```
Algorithm

`convhull` is based on Qhull [2]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also

`convhulln`, `delaunay`, `plot`, `polyarea`, `voronoi`

Reference


Purpose

N-dimensional convex hull

Syntax

K = convhulln(X)
K = convhulln(X, options)
[K,v] = convhulln(...)

Description

K = convhulln(X) returns the indices K of the points in X that comprise the facets of the convex hull of X. X is an m-by-n array representing m points in N-dimensional space. If the convex hull has p facets then K is p-by-n.

convhulln uses Qhull.

K = convhulln(X, options) specifies a cell array of strings options to be used as options in Qhull. The default options are:

• {'Qt'} for 2-, 3-. and 4-dimensional input
• {'Qt', 'Qx'} for 5-dimensional input and higher.

If options is [], the default options are used. If options is {}, no options are used, not even the default. For more information on Qhull and its options, see http://www.qhull.org/.

[K, v] = convhulln(...) also returns the volume v of the convex hull.

Visualization

Plotting the output of convhulln depends on the value of n:

• For n = 2, use plot as you would for convhull.
• For n = 3, you can use trisurf to plot the output. The calling sequence is
  K = convhulln(X);
  trisurf(K,X(:,1),X(:,2),X(:,3))

  For more control over the color of the facets, use patch to plot the output. For an example, see “Tessellation and Interpolation of Scattered Data in Higher Dimensions” in the MATLAB documentation.
• You cannot plot convhulln output for n > 3.

Algorithm

convhulln is based on Qhull [2]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.
**convhulln**

**See Also**
convhull, delaunayn, dsearchn, tsearchn, voronoin

**Reference**


Purpose  
N-dimensional convolution

Syntax  
C = convn(A,B)
C = convn(A,B,'shape')

Description  
C = convn(A,B) computes the N-dimensional convolution of the arrays A and B. The size of the result is size(A)+size(B)-1.

C = convn(A,B,'shape') returns a subsection of the N-dimensional convolution, as specified by the shape parameter:

'full'  Returns the full N-dimensional convolution (default).
'same'  Returns the central part of the result that is the same size as A.
'valid'  Returns only those parts of the convolution that can be computed without assuming that the array A is zero-padded. The size of the result is
max(size(A)*size(B) + 1, 0)

See Also  
conv, conv2
copyfile

**Purpose**
Copy file or directory

**Graphical Interface**
As an alternative to the `copyfile` function, use the Current Directory browser. Select the files and then select copy and paste commands from the Edit menu.

**Syntax**
copyfile('source','destination')
copyfile('source','destination','f')
[status,message,messageid] = copyfile('source','destination','f')

**Description**
copyfile('source','destination') copies the file or directory, source (and all its contents) to the file or directory, destination, where source and destination are the absolute or relative pathnames for the directory or file. If source is a directory, destination cannot be a file. If source is a directory, copyfile copies the contents of source, not the directory itself. To rename a file or directory when copying it, make destination a different name than source. If destination already exists, copyfile replaces it without warning. Use the wildcard * at the end of source to copy all matching files. Note that the read-only and archive attributes of source are not preserved in destination.

copyfile('source','destination','f') copies source to destination, regardless of the read-only attribute of destination.

[status,message,messageid] = copyfile('source','destination','f') copies source to destination, returning the status, a message, and the MATLAB error message ID (see error and lasterr). Here, status is 1 for success and 0 for error. Only one output argument is required and the f input argument is optional.

The * wildcard in a path string is supported. Current behavior of copyfile differs between UNIX and Windows when using the wildcard * or copying directories.

**Examples**

**Copy File in Current Directory, Assigning a New Name to It**
To make a copy of a file `myfun.m` in the current directory, assigning it the name `myfun2.m`, type

```matlab
copyfile('myfun.m','myfun2.m')
```
Copy File to Another Directory
To copy `myfun.m` to the directory `d:/work/myfiles`, keeping the same filename, type:

```
copyfile('myfun.m','d:/work/myfiles')
```

Copy All Matching Files by Using a Wildcard
To copy all files in the directory `myfiles` whose names begin with `my` to the directory `newprojects`, where `newprojects` is at the same level as the current directory, type:

```
copyfile('myfiles/my*','../newprojects')
```

Copy Directory and Return Status
In this example, all files and subdirectories in the current directory's `myfiles` directory are copied to the directory `d:/work/myfiles`. Note that before running the `copyfile` function, `d:/work` does not contain the directory `myfiles`. It is created because `myfiles` is appended to `destination` in the `copyfile` function:

```
[s,mess,messid]=copyfile('myfiles','d:/work/myfiles')
```

```
s =
    1

mess =
    ..

messid =
    ..
```

The message returned indicates that `copyfile` was successful.

Copy File to Read-Only Directory
Copy `myfile.m` from the current directory to `d:/work/restricted`, where `restricted` is a read-only directory:

```
copyfile('myfile.m','d:/work/restricted','f')
```

After the copy, `myfile.m` exists in `d:/work/restricted`.

See Also
`cd`, `delete`, `dir`, `fileattrib`, `filebrowser`, `fileparts`, `mkdir`, `movefile`, `rmdir`
copyobj

**Purpose**
Copy graphics objects and their descendants

**Syntax**
new_handle = copyobj(h,p)

**Description**
copyobj creates copies of graphics objects. The copies are identical to the original objects except the copies have different values for their Parent property and a new handle. The new parent must be appropriate for the copied object (e.g., you can copy a line object only to another axes object).

new_handle = copyobj(h,p) copies one or more graphics objects identified by h and returns the handle of the new object or a vector of handles to new objects. The new graphics objects are children of the graphics objects specified by p.

**Remarks**
h and p can be scalars or vectors. When both are vectors, they must be the same length, and the output argument, new_handle, is a vector of the same length. In this case, new_handle(i) is a copy of h(i) with its Parent property set to p(i).

When h is a scalar and p is a vector, h is copied once to each of the parents in p. Each new_handle(i) is a copy of h with its Parent property set to p(i), and length(new_handle) equals length(p).

When h is a vector and p is a scalar, each new_handle(i) is a copy of h(i) with its Parent property set to p. The length of new_handle equals length(h).

Graphics objects are arranged as a hierarchy. See Handle Graphics Objects for more information.

**Examples**
Copy a surface to a new axes within a different figure.

```matlab
h = surf(peaks);
colormap hot
figure % Create a new figure
axes % Create an axes object in the figure
new_handle = copyobj(h,gca);
colormap hot
view(3)
grid on
```

Note that while the surface is copied, the colormap (figure property), view, and grid (axes properties) are not copies.
See Also

findobj, gcf, gca, gco, get, set

Parent property for all graphics objects

“Finding and Identifying Graphics Objects” for related functions
Purpose
Correlation coefficients

Syntax
R = corrcoef(X)
R = corrcoef(x,y)
[R,P]=corrcoef(...)  
[R,P,RLO,RUP]=corrcoef(...)  
[...]=corrcoef(...,'param1',val1,'param2',val2,...)

Description
R = corrcoef(X) returns a matrix R of correlation coefficients calculated from an input matrix X whose rows are observations and whose columns are variables. The matrix R = corrcoef(X) is related to the covariance matrix C = cov(X) by

\[ R(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}} \]

corrcoef(X) is the zeroth lag of the covariance function, that is, the zeroth lag of xcov(x, 'coeff') packed into a square array.

R = corrcoef(x,y) where x and y are column vectors is the same as corrcoef([x y]).

[R,P]=corrcoef(...) also returns P, a matrix of p-values for testing the hypothesis of no correlation. Each p-value is the probability of getting a correlation as large as the observed value by random chance, when the true correlation is zero. If \( P(i,j) \) is small, say less than 0.05, then the correlation \( R(i,j) \) is significant.

[R,P,RLO,RUP]=corrcoef(...) also returns matrices RLO and RUP, of the same size as R, containing lower and upper bounds for a 95% confidence interval for each coefficient.

[...]=corrcoef(...,'param1',val1,'param2',val2,...) specifies additional parameters and their values. Valid parameters are the following.
The p-value is computed by transforming the correlation to create a t statistic having \( n-2 \) degrees of freedom, where \( n \) is the number of rows of \( X \). The confidence bounds are based on an asymptotic normal distribution of \( 0.5 \log((1+R)/(1-R)) \), with an approximate variance equal to \( 1/(n-3) \). These bounds are accurate for large samples when \( X \) has a multivariate normal distribution. The 'pairwise' option can produce an \( R \) matrix that is not positive definite.

**Examples**

Generate random data having correlation between column 4 and the other columns.

```matlab
x = randn(30,4); % Uncorrelated data
x(:,4) = sum(x,2); % Introduce correlation.
[r,p] = corrcoef(x) % Compute sample correlation and p-values.
[i,j] = find(p<0.05); % Find significant correlations.
[i,j] % Display their (row,col) indices.
```

```plaintext
r =
 1.0000  -0.3566   0.1929   0.3457
-0.3566   1.0000  -0.1429   0.4461
 0.1929  -0.1429   1.0000   0.5183
 0.3457   0.4461   0.5183   1.0000

p =
 1.0000   0.0531   0.3072   0.0613
 0.0531   1.0000   0.4511   0.0135
 0.3072   0.4511   1.0000   0.0033
 0.0613   0.0135   0.0033   1.0000

ans =
 4     2
 4     3
 2     4
```
See Also

cov, mean, std

corr, xcov in the Signal Processing Toolbox
**Purpose**
Cosine of an argument in radians

**Syntax**
\[ Y = \cos(X) \]

**Description**
The \( \cos \) function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

\[ Y = \cos(X) \] returns the circular cosine for each element of \( X \).

**Examples**
Graph the cosine function over the domain \(-\pi \leq x \leq \pi\).

\[ x = -\pi:0.01:\pi; \]
\[ plot(x,\cos(x)), \text{grid on} \]

![Graph of cosine function](image)

The expression \( \cos(\pi/2) \) is not exactly zero but 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 cosine can be defined as

\[
\cos(x + iy) = \cos(x)\cosh(y) - i\sin(x)\sinh(y)
\]

\[
\cos(z) = \frac{e^{iz} + e^{-iz}}{2}
\]
**COS**

**Algorithm**

`cos` 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**

`acos`, `acosh`, `cosd`, `cosh`
Purpose
Cosine of an argument in degrees

Syntax
Y = cosd(X)

Description
Y = cosd(X) is the cosine of the elements of X, expressed in degrees. For odd integers n, \(\text{cosd}(n*90)\) is exactly zero, whereas \(\cos(n*pi/2)\) reflects the accuracy of the floating point value of pi.

See Also
acosd, cos
**Purpose**
Hyperbolic cosine

**Syntax**
\[ Y = \cosh(X) \]

**Description**
The \cosh function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

\[ Y = \cosh(X) \] returns the hyperbolic cosine for each element of \( X \).

**Examples**
Graph the hyperbolic cosine function over the domain \(-5 \leq x \leq 5\).

\[ x = -5:0.01:5; \]
\[ \text{plot}(x,\cosh(x)), \text{grid on} \]

**Definition**
The hyperbolic cosine can be defined as

\[ \cosh(z) = \frac{e^{2z} + e^{-2z}}{2} \]

**Algorithm**
\cosh 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  acos, acosh, cos
**cot**

**Purpose**  
Cotangent of an argument in radians

**Syntax**  
\[ Y = \cot(X) \]

**Description**  
The \( \cot \) function operates element-wise on arrays. The function’s domains and ranges include complex values. All angles are in radians.

\[ Y = \cot(X) \]  
returns the cotangent for each element of \( X \).

**Examples**  
Graph the cotangent the domains \(-\pi < x < 0\) and \(0 < x < \pi\).

\[
x1 = -\pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:\pi-0.01;
plot(x1,cot(x1),x2,cot(x2)), grid on
\]

**Definition**  
The cotangent can be defined as

\[
\cot(z) = \frac{1}{\tan(z)}
\]

**Algorithm**  
\( \cot \) 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  acot, acoth, cotd, coth
**Purpose**
Cotangent of an argument in degrees

**Syntax**
\[ Y = \text{cotd}(X) \]

**Description**
\[ Y = \text{cotd}(X) \] is the cotangent of the elements of \( X \), expressed in degrees. For integers \( n \), \( \text{cotd}(n\times180) \) is infinite, whereas \( \text{cot}(n\times\pi) \) is large but finite, reflecting the accuracy of the floating point value of \( \pi \).

**See Also**
acotd, cot
Purpose

Hyperbolic cotangent

Syntax

\[ Y = \coth(X) \]

Description

The \( \coth \) function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

\[ Y = \coth(X) \] returns the hyperbolic cotangent for each element of \( X \).

Examples

Graph the hyperbolic cotangent over the domains \(-\pi < x < 0\) and \(0 < x < \pi\).

\[
\begin{align*}
x1 &= -\pi + 0.01:0.01:-0.01; \\
x2 &= 0.01:0.01:\pi-0.01; \\
plot(x1,\coth(x1),x2,\coth(x2)), \text{ grid on}
\end{align*}
\]

Definition

The hyperbolic cotangent can be defined as

\[ \coth(z) = \frac{1}{\tanh(z)} \]

Algorithm

coth 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.
coth

See Also  acot, acoth, cot
**Purpose**

Covariance matrix

**Syntax**

\[ C = \text{cov}(X) \]
\[ C = \text{cov}(x,y) \]

**Description**

\[ C = \text{cov}(x) \] where \( x \) is a vector returns the variance of the vector elements. For matrices where each row is an observation and each column a variable, \( \text{cov}(x) \) is the covariance matrix. \( \text{diag}(\text{cov}(x)) \) is a vector of variances for each column, and \( \sqrt{\text{diag}(\text{cov}(x))} \) is a vector of standard deviations.

\[ C = \text{cov}(x,y), \] where \( x \) and \( y \) are column vectors of equal length, is equivalent to \( \text{cov}([x \ y]) \).

**Remarks**

cov removes the mean from each column before calculating the result.

The covariance function is defined as

\[ \text{cov}(x_1,x_2) = E[(x_1 - \mu_1)(x_2 - \mu_2)] \]

where \( E \) is the mathematical expectation and \( \mu_i = Ex_i \).

**Examples**

Consider \( A = [-1 1 2; -2 3 1; 4 0 3] \). To obtain a vector of variances for each column of \( A \):

\[ v = \text{diag}(\text{cov}(A))' \]
\[ v = \begin{bmatrix} 10.3333 & 2.3333 & 1.0000 \end{bmatrix} \]

Compare vector \( v \) with covariance matrix \( C \):

\[ C = \begin{bmatrix} 10.3333 & -4.1667 & 3.0000 \\ -4.1667 & 2.3333 & -1.5000 \\ 3.0000 & -1.5000 & 1.0000 \end{bmatrix} \]

The diagonal elements \( C(i,i) \) represent the variances for the columns of \( A \). The off-diagonal elements \( C(i,j) \) represent the covariances of columns \( i \) and \( j \).

**See Also**

corrcoef, mean, std

xcov, xcorr in the Signal Processing Toolbox
cplxpair

Purpose
Sort complex numbers into complex conjugate pairs

Syntax
B = cplxpair(A)
B = cplxpair(A,tol)
B = cplxpair(A,[],dim)
B = cplxpair(A,tol,dim)

Description
B = cplxpair(A) sorts the elements along different dimensions of a complex
array, grouping together complex conjugate pairs.
The conjugate pairs are ordered by increasing real part. Within a pair, the
element with negative imaginary part comes first. The purely real values are
returned following all the complex pairs. The complex conjugate pairs
are forced to be exact complex conjugates. A default tolerance of \(100*\text{eps}\) relative
to \(\text{abs}(A(i))\) determines which numbers are real and which elements are
paired complex conjugates.
If \(A\) is a vector, \texttt{cplxpair(A)} returns \(A\) with complex conjugate pairs grouped
together.
If \(A\) is a matrix, \texttt{cplxpair(A)} returns \(A\) with its columns sorted and complex
conjugates paired.
If \(A\) is a multidimensional array, \texttt{cplxpair(A)} treats the values along the first
non-singleton dimension as vectors, returning an array of sorted elements.
B = cplxpair(A,tol) overrides the default tolerance.
B = cplxpair(A,[],dim) sorts \(A\) along the dimension specified by scalar \(\text{dim}\).
B = cplxpair(A,tol,dim) sorts \(A\) along the specified dimension and overrides
the default tolerance.

Diagnostics
If there are an odd number of complex numbers, or if the complex numbers
cannot be grouped into complex conjugate pairs within the tolerance, \texttt{cplxpair}
generates the error message
Complex numbers can't be paired.
Purpose

Elapsed CPU time

Syntax

cputime

Description

cputime returns the total CPU time (in seconds) used by MATLAB from the time it was started. This number can overflow the internal representation and wrap around.

Examples

The following code returns the CPU time used to run surf(peaks(40)).

```
t = cputime; surf(peaks(40)); e = cputime-t
```

e =

0.4667

See Also

clock, etime, tic, toc
createClassFromWsdl

**Purpose**
Creates MATLAB classes from Web Services Description Language (WSDL)

**Syntax**
createClassFromWsdl('source')

**Description**
createClassFromWsdl('source') creates MATLAB classes based on a WSDL application programming interface (API). The source argument specifies a URL or file path to a WSDL API, which defines web service methods, arguments, and transactions.

Based on the WSDL API, the createClassFromWsdl function creates a new folder in the current directory. The folder contains an M-file for each web service method. In addition, two default M-files are created that display method results (display.m) and that initialize the web service MATLAB object (servicename.m).

For example, if myWebService offers two methods (method1 and method2), the createClassFromWsdl function creates:

- @myWebService folder in the current directory
- method1.m — M-file for method1
- method2.m — M-file for method2
- display.m — Default M-file for display method
- myWebService.m — Default M-file for the myWebService MATLAB object

**Remarks**
For more information about WSDL and web services, see the following resources:

- World Wide Web Consortium (W3C) WSDL specification
- W3C SOAP specification
- XMethods.net

**Example**
The following example calls a web service that returns the book price for an International Standard Bibliographic Number (ISBN).

```matlab
% The createClassFromWsdl function takes the WSDL URL as an argument.
createClassFromWsdl('http://www.xmethods.net/sd/2001/BNQuoteService.wsdl');
bq = bnquoteservice;
```

2-490
% getQuote is the web service method. The first argument, bq, is an instance of the bnquoteservice class. The second argument, 0735712719, is an ISBN number.
getprice(bq, '0735712719');
cross

Purpose
Vector cross product

Syntax
C = cross(A,B)
C = cross(A,B,dim)

Description
C = cross(A,B) returns the cross product of the vectors A and B. That is, 
\[ C = A \times B \]. A and B must be 3-element vectors. If A and B are multidimensional 
arrays, cross returns the cross product of A and B along the first dimension of 
length 3.

C = cross(A,B,dim) where A and B are multidimensional arrays, returns the 
cross product of A and B in dimension dim. A and B must have the same size, 
and both size(A,dim) and size(B,dim) must be 3.

Remarks
To perform a dot (scalar) product of two vectors of the same size, use 
c = dot(a,b).

Examples
The cross and dot products of two vectors are calculated as shown:

\[
a = [1 \ 2 \ 3];
b = [4 \ 5 \ 6];
c = cross(a,b)
\]

\[
c =
\begin{array}{ccc}
-3 & 6 & -3 \\
\end{array}
\]

d = dot(a,b)

d =
32

See Also
dot
Purpose

Cosecant of an argument in radians

Syntax

\[ Y = \csc(x) \]

Description

The \( \csc \) function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

\[ Y = \csc(x) \] returns the cosecant for each element of \( x \).

Examples

Graph the cosecant over the domains \(-\pi < x < 0 \) and \( 0 < x < \pi \).

\[
\begin{align*}
x1 &= \text{pi}+0.01:0.01:\text{pi}-0.01; \\
x2 &= 0.01:0.01:pi-0.01; \\
\text{plot}(x1,\csc(x1),x2,\csc(x2)), \text{grid on}
\end{align*}
\]

Definition

The cosecant can be defined as

\[ \csc(z) = \frac{1}{\sin(z)} \]

Algorithm

\( \csc \) 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  
acs, acsch, cscd, csch
Purpose  Cosecant of an argument in degrees

Syntax  \( Y = \text{cscd}(X) \)

Description  \( Y = \text{cscd}(X) \) is the cosecant of the elements of \( X \), expressed in degrees. For integers \( n \), \( \text{cscd}(n*180) \) is infinite, whereas \( \text{csc}(n*\pi) \) is large but finite, reflecting the accuracy of the floating point value of \( \pi \).

See Also  acscd, csc
csch

Purpose
Hyperbolic cosecant

Syntax
Y = csch(x)

Description
The csch function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Y = csch(x) returns the hyperbolic cosecant for each element of x.

Examples
Graph the hyperbolic cosecant over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```matlab
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csch(x1),x2,csch(x2)), grid on
```

Definition
The hyperbolic cosecant can be defined as

$$\text{csch}(z) = \frac{1}{\sinh(z)}$$

Algorithm
csch 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  
acsc, acsch, csc
csvread

Purpose
Read a comma-separated value file

Syntax
M = csvread('filename')
M = csvread('filename', row, col)
M = csvread('filename', row, col, range)

Description
M = csvread('filename') reads a comma-separated value formatted file, filename. The result is returned in M. The file can only contain numeric values.

M = csvread('filename', row, col) reads data from the comma-separated value formatted file starting at the specified row and column. The row and column arguments are zero based, so that row=0 and col=0 specify the first value in the file.

M = csvread('filename', row, col, range) reads only the range specified. Specify range using the notation [R1 C1 R2 C2] where (R1,C1) is the upper left corner of the data to be read and (R2,C2) is the lower right corner. You can also specify the range using spreadsheet notation, as in range = 'A1..B7'.

Remarks
csvread fills empty delimited fields with zero. Data files having lines that end with a nonspace delimeter, such as a semicolon, produce a result that has an additional last column of zeros.
csvread 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;-&lt;real&gt;-&lt;imag&gt;i</td>
<td>j</td>
</tr>
<tr>
<td>&lt;-&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.

Examples
Given the file csvlist.dat that contains the comma-separated values

<table>
<thead>
<tr>
<th>02, 04, 06, 08, 10, 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>03, 06, 09, 12, 15, 18</td>
</tr>
</tbody>
</table>
05, 10, 15, 20, 25, 30
07, 14, 21, 28, 35, 42
11, 22, 33, 44, 55, 66

To read the entire file, use

```matlab
csvread('csvlist.dat')
```

```
ans =
     2     4     6     8    10    12
     3     6     9    12    15    18
     5    10    15    20    25    30
     7    14    21    28    35    42
    11    22    33    44    55    66
```

To read the matrix starting with zero-based row 2, column 0, and assign it to the variable m,

```matlab
m = csvread('csvlist.dat', 2, 0)
```

```
m =
    5    10    15    20    25    30
    7    14    21    28    35    42
   11    22    33    44    55    66
```

To read the matrix bounded by zero-based (2,0) and (3,3) and assign it to m,

```matlab
m = csvread('csvlist.dat', 2, 0, [2,0,3,3])
```

```
m =
    5    10    15    20
    7    14    21    28
```

**See Also**
csvwrite, dlmread, textscan, wk1read, file formats, importdata, uimport
csvwrite

Purpose
Write a comma-separated value file

Syntax
csvwrite('filename',M)
csvwrite('filename',M,row,col)

Description
csvwrite('filename',M) writes matrix M into filename as comma-separated values.
csvwrite('filename',M,row,col) writes matrix M into filename starting at the specified row and column offset. The row and column arguments are zero based, so that row=0 and C=0 specify the first value in the file.

Examples
The following example creates a comma-separated value file from the matrix m.

\[
m = \begin{bmatrix}
3 & 6 & 9 & 12 & 15 \\
5 & 10 & 15 & 20 & 25 \\
7 & 14 & 21 & 28 & 35 \\
11 & 22 & 33 & 44 & 55
\end{bmatrix}
\]

csvwrite('csvlist.dat',m)
type csvlist.dat

3,6,9,12,15
5,10,15,20,25
7,14,21,28,35
11,22,33,44,55

The next example writes the matrix to the file, starting at a column offset of 2.

csvwrite('csvlist.dat',m,0,2)
type csvlist.dat

,,3,6,9,12,15
,,5,10,15,20,25
,,7,14,21,28,35
,,11,22,33,44,55

See Also
csvread, dlmwrite, textread, wk1write, file formats, importdata, uimport
Purpose
Cumulative product

Syntax
B = cumprod(A)
B = cumprod(A,dim)

Description
B = cumprod(A) returns the cumulative product along different dimensions of an array.

If A is a vector, cumprod(A) returns a vector containing the cumulative product of the elements of A.

If A is a matrix, cumprod(A) returns a matrix the same size as A containing the cumulative products for each column of A.

If A is a multidimensional array, cumprod(A) works on the first nonsingleton dimension.

B = cumprod(A,dim) returns the cumulative product of the elements along the dimension of A specified by scalar dim. For example, cumprod(A,1) increments the first (row) index, thus working along the rows of A.

Examples

cumprod(1:5)
ans =
1  2  6  24  120

A = [1 2 3; 4 5 6];
cumprod(A)
ans =
1  2  3
4 10 18

cumprod(A,2)
ans =
1  2  6
4 20 120

See Also
cumsum, prod, sum
cumsum

Purpose
Cumulative sum

Syntax
B = cumsum(A)
B = cumsum(A,dim)

Description
B = cumsum(A) returns the cumulative sum along different dimensions of an array.
If A is a vector, cumsum(A) returns a vector containing the cumulative sum of the elements of A.
If A is a matrix, cumsum(A) returns a matrix the same size as A containing the cumulative sums for each column of A.
If A is a multidimensional array, cumsum(A) works on the first nonsingleton dimension.
B = cumsum(A,dim) returns the cumulative sum of the elements along the dimension of A specified by scalar dim. For example, cumsum(A,1) works across the first dimension (the rows).

Examples
```
cumsum(1:5)
ans =
    1  3  6  10  15

A = [1 2 3; 4 5 6];
cumsum(A)
ans =
    1  2  3
    5  7  9

cumsum(A,2)
ans =
    1  3  6
    4  9  15
```

See Also
cumprod, prod, sum
Purpose
Cumulative trapezoidal numerical integration

Syntax
Z = cumtrapz(Y)
Z = cumtrapz(X,Y)
Z = cumtrapz(... dim)

Description
Z = cumtrapz(Y) computes an approximation of the cumulative integral of Y via the trapezoidal method with unit spacing. To compute the integral with other than unit spacing, multiply Z by the spacing increment.

For vectors, cumtrapz(Y) is a vector containing the cumulative integral of Y.

For matrices, cumtrapz(Y) is a matrix the same size as Y with the cumulative integral over each column.

For multidimensional arrays, cumtrapz(Y) works across the first nonsingleton dimension.

Z = cumtrapz(X,Y) computes the cumulative integral of Y with respect to X using trapezoidal integration. X and Y must be vectors of the same length, or X must be a column vector and Y an array whose first nonsingleton dimension is length(X). cumtrapz operates across this dimension.

If X is a column vector and Y an array whose first nonsingleton dimension is length(X), cumtrapz(X,Y) operates across this dimension.

Z = cumtrapz(X,Y,dim) or cumtrapz(Y,DIM) integrates across the dimension of Y specified by scalar dim. The length of X must be the same as size(Y,dim).

Example
Y = [0 1 2; 3 4 5];
cumtrapz(Y,1)
an =
 0  0  0
 1.5000  2.5000  3.5000
cumtrapz(Y,2)
an =
 0  0.5000  2.0000
 0  3.5000  8.0000
cumtrapz

See Also

cumsum, trapz
Purpose

Computes the curl and angular velocity of a vector field

Syntax

```
[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)
[curlx,curly,curlz,cav] = curl(U,V,W)
[curlz,cav] = curl(X,Y,U,V)
[curlz,cav] = curl(U,V)
[curlx,curly,curlz] = curl(...), [curlx,curly] = curl(...)  
cav = curl(...)
```

Description

- 
```
[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W) computes the curl and angular velocity perpendicular to the flow (in radians per time unit) of a 3-D vector field \( U, V, W \). 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`).
```

- 
```
[curlx,curly,curlz,cav] = curl(U,V,W) assumes \( X, Y, \) and \( Z \) are determined by the expression
```
```
\( [X Y Z] = \text{meshgrid}(1:n,1:m,1:p) \)
```
```
where \([m,n,p] = \text{size}(U)\).
```

- 
```
[curlz,cav] = curl(X,Y,U,V) computes the curl z-component and the angular velocity perpendicular to z (in radians per time unit) of a 2-D vector field \( U, V \). The arrays \( X, Y \) define the coordinates for \( U, V \) and must be monotonic and 2-D plaid (as if produced by `meshgrid`).
```

- 
```
[curlz,cav] = curl(U,V) assumes \( X \) and \( Y \) are determined by the expression
```
```
\( [X Y] = \text{meshgrid}(1:n,1:m) \)
```
```
where \([m,n] = \text{size}(U)\).
```

- 
```
[curlx,curly,curlz] = curl(...), [curlx,curly] = curl(...) returns only the curl.
```

- 
```
cav = curl(...) returns only the curl angular velocity.
```

Examples

This example uses colored slice planes to display the curl angular velocity at specified locations in the vector field.
This example views the curl angular velocity in one plane of the volume and plots the velocity vectors (quiver) in the same plane.

```matlab
load wind
k = 4;
x = x(:,:,k); y = y(:,:,k); u = u(:,:,k); v = v(:,:,k);
cav = curl(x,y,u,v);
pcolor(x,y,cav); shading interp
hold on;
quiver(x,y,u,v,'y')
hold off
colormap copper
```
See Also

streamribbon, divergence

“Volume Visualization” for related functions
Displaying Curl with Stream Ribbons for another example
**customverctrl**

**Purpose**
Allow custom source control system

**Syntax**
customverctrl(filename, arguments)

**Description**
This function is supplied for customers who want to integrate a version control system that is not supported with MATLAB. This function must conform to the structure of one of the supported version control systems, for example RCS. See the files clearcase.m, pvcs.m, rcs.m, and sourcesafe.m in $matlabroot\toolbox\matlab\verctrl as examples.

**See Also**
checkin, checkout, cmopts, undocheckout
cylinder

Purpose

Generate cylinder

Syntax

\[
\begin{align*}
[X,Y,Z] &= \text{cylinder} \\
[X,Y,Z] &= \text{cylinder}(r) \\
[X,Y,Z] &= \text{cylinder}(r,n) \\
\text{cylinder}(\text{axes} \_\text{handle},...) \\
\text{cylinder}(...) 
\end{align*}
\]

Description

cylinder generates \(x\)-, \(y\)-, and \(z\)-coordinates of a unit cylinder. You can draw the cylindrical object using \textit{surf} or \textit{mesh}, or draw it immediately by not providing output arguments.

\([X,Y,Z] = \text{cylinder}\) returns the \(x\)-, \(y\)-, and \(z\)-coordinates of a cylinder with a radius equal to 1. The cylinder has 20 equally spaced points around its circumference.

\([X,Y,Z] = \text{cylinder}(r)\) returns the \(x\)-, \(y\)-, and \(z\)-coordinates of a cylinder using \(r\) to define a profile curve. \text{cylinder} treats each element in \(r\) as a radius at equally spaced heights along the unit height of the cylinder. The cylinder has 20 equally spaced points around its circumference.

\([X,Y,Z] = \text{cylinder}(r,n)\) returns the \(x\)-, \(y\)-, and \(z\)-coordinates of a cylinder based on the profile curve defined by vector \(r\). The cylinder has \(n\) equally spaced points around its circumference.

\text{cylinder}(\text{axes} \_\text{handle},...)\) plots into the axes with handle \text{axes} \_\text{handle} instead of the current axes (\textit{gca}).

\text{cylinder}(...), with no output arguments, plots the cylinder using \textit{surf}.

Remarks

cylinder treats its first argument as a profile curve. The resulting surface graphics object is generated by rotating the curve about the \(x\)-axis, and then aligning it with the \(z\)-axis.

Examples

Create a cylinder with randomly colored faces.

\begin{verbatim}
cylinder
axis square
h = findobj('Type','surface');
\end{verbatim}
Generate a cylinder defined by the profile function $2 + \sin(t)$.

```matlab
t = 0:pi/10:2*pi;
[X,Y,Z] = cylinder(2+cos(t));
surf(X,Y,Z)
axis square
```

set(h,'CData',rand(size(get(h,'CData'))))
Purpose
Set or query the axes data aspect ratio

Syntax

daspect

daspect([aspect_ratio])

daspect('mode')

daspect('auto')

daspect('manual')

daspect(axes_handle,...)

Description
The data aspect ratio determines the relative scaling of the data units along the x-, y-, and z-axes.

daspect with no arguments returns the data aspect ratio of the current axes.

daspect([aspect_ratio]) sets the data 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-axis scaling (e.g., [1 1 3] means one unit in x is equal in length to one unit in y and three units in z).

daspect('mode') returns the current value of the data aspect ratio mode, which can be either auto (the default) or manual. See Remarks.

daspect('auto') sets the data aspect ratio mode to auto.

daspect('manual') sets the data aspect ratio mode to manual.

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

Remarks
daspect sets or queries values of the axes object DataAspectRatio and DataAspectRatioMode properties.

When the data aspect ratio mode is auto, MATLAB adjusts the data aspect ratio so that each axis spans the space available in the figure window. If you are displaying a representation of a real-life object, you should set the data aspect ratio to [1 1 1] to produce the correct proportions.

Setting a value for data aspect ratio or setting the data aspect ratio mode to manual disables the MATLAB stretch-to-fill feature (stretching of the axes to
This means setting the data aspect ratio to a value, including its current value,

    daspect(daspect)

can cause a change in the way the graphs look. See the Remarks section of the axes description for more information.

Examples

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

    [x,y] = meshgrid([-2:.2:2]);
    z = x.*exp(-x.^2 - y.^2);
    surf(x,y,z)

Querying the data aspect ratio shows how MATLAB has drawn the surface.

    daspect
    ans =
    4   4   1

Setting the data aspect ratio to \([1 1 1]\) produces a surface plot with equal scaling along each axis.

    daspect([1 1 1])
See Also

axis, pbaspect, xlim, ylim, zlim

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

“Setting the Aspect Ratio and Axis Limits” for related functions

Axes Aspect Ratio for more information
datacursormode

**Purpose**
Enable/disable interactive data cursor mode

**Syntax**
datacursormode on
datacursormode off
datacursormode
datacursormode(figure_handle,...)
dcm_obj = datacursormode(figure_handle)

**Description**
datacursormode on enables data cursor mode on the current figure.
datacursormode off disables data cursor mode on the current figure.
datacursormode toggles data cursor mode on the current figure.
datacursormode(figure_handle,...) enables or disables data cursor mode on the specified figure.
dcm_obj = datacursormode(figure_handle) returns the figure's data cursor mode object, which enables you to customize the data cursor. See “Data Cursor Mode Object”.

**Data Cursor Mode Object**
The data cursor mode object has properties that enable you to controls certain aspects of the data cursor. You can use the set and get commands and the returned object (dcm_obj in the above syntax) to set and query property values.

**Data Cursor Mode Properties**

- **Enabled**: on | off
  Specifies whether this mode is currently enabled on the figure.

- **SnapToDataVertex**: on | off
  Specifies whether the data cursor snaps to the nearest data value or is located at the actual pointer position.

- **DisplayStyle**: datatip | window
  Determines how the data is displayed.
  - **datatip**: displays cursor information in a yellow text box next to a marker indicating the actual data point being displayed.

2-514
window displays cursor information in a floating window within the figure.

**Updatefcn**  
function handle

This property references a function that customizes the text appearing in the data cursor. The function handle must reference a function that has two implicit arguments (these arguments are automatically pass to the function by MATLAB when the function executes). For example, the following function definition line uses the required arguments:

```matlab
function output_txt = myfunction(obj,event_obj)
% obj Currently not used (empty)
% event_obj Handle to event object
% output_txt Data cursor text string (string or cell array of strings).
```

event_obj is an object having the following read-only properties.

- **Target** – Handle of the object the data cursor is referencing (the object on which the user clicked).
- **Position** – An array specifying the x, y, (and z for 3-D graphs) coordinates of the cursor.

You can query these properties within your function. For example,

```matlab
pos = get(event_obj,'Position');
```

returns the coordinates of the cursor.

See Function Handles for more information on creating a function handle.

See “Change Data Cursor Text” for an example.

**Data Cursor Method**

You can use the `getCursorInfo` function with the data cursor mode object (dcm_obj in the above syntax) to obtain information about the data cursor. For example,

```matlab
info_struct = getCursorInfo(dcm_obj);
```

returns a vector of structures, one for each data cursor on the graph. Each structure has the following fields:

- **Target** — The handle of the graphics object containing the data point.
• Position — An array specifying the x, y, (and z) coordinates of the cursor.

Line and lineseries objects have an additional field:
• DataIndex — A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

Examples
This example creates a plot and enables data cursor mode from the command line.

```matlab
surf(peaks)
datacursormode on
% Click mouse on surface to display data cursor
```

Setting Data Cursor Mode Options
This example enables data cursor mode on the current figure and sets data cursor mode options. The following statements

• Create a graph
• Toggle data cursor mode to on
• Save the data cursor mode object to specify options and get the handle of the line to which the datatip is attached.

```matlab
fig = figure;
z = peaks;
plot(z(:,30:35))
dcm_obj = datacursormode(fig);
set(dcm_obj,'DisplayStyle','datatip','SnapToDataVertex','off')
% Click on line to place datatip

c_info = getCursorInfo(dcm_obj);
set(c_info.Target,'LineWidth',2) % Make selected line wider
```
Change Data Cursor Text

This example shows you how to customize the text that is displayed by the data cursor. Suppose you want to replace the text displayed in the datatip and data window with "Time:" and "Amplitude:"

```matlab
function doc_datacursormode
    fig = figure;
    a = -16; t = 0:60;
    plot(t,sin(a*t))
    dcm_obj = datacursormode(fig);
    set(dcm_obj,'UpdateFcn',@myupdatefcn)

    % Click on line to select data point

    function txt = myupdatefcn(empt,event_obj)
        pos = get(event_obj,'Position');
        txt = {{'Time: ',num2str(pos(1))},
               {'Amplitude: ',num2str(pos(2))}};
```
datatipinfo

Purpose
Produce short description of input variable

Syntax
datatipinfo(var)

Description
datatipinfo(var) displays a short description of a variable, similar to what is displayed in a datatip in the MATLAB debugger.

Examples
Get datatip information for a 5-by-5 matrix:

A = rand(5);
datatipinfo(A)

A: 5x5 double =
0.4445   0.3567   0.7458   0.0767   0.4400
0.7962   0.6575   0.3918   0.8289   0.9746
0.5641   0.9808   0.0265   0.4838   0.6722
0.9099   0.9653   0.2508   0.4859   0.4054
0.2857   0.5198   0.7383   0.9301   0.9604

Get datatip information for a 50-by-50 matrix. For this larger matrix, datatipinfo displays just the size and data type:

A = rand(50);
datatipinfo(A)

A: 50x50 double

Also for multidimensional matrices, datatipinfo displays just the size and data type:

A = rand(5);
A(:,:,2) = A(:,:,1);
datatipinfo(A)

A: 5x5x2 double

See Also
debug
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Current date string</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>str = date</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>str = date returns a string containing the date in dd-mmm-yyyy format.</td>
</tr>
<tr>
<td><strong>See Also</strong></td>
<td>clock, datenum, now</td>
</tr>
</tbody>
</table>
Purpose

Convert to serial date number

Syntax

\[ N = \text{datenum}(DT) \]
\[ N = \text{datenum}(DT, P) \]
\[ N = \text{datenum}(DT, F) \]
\[ N = \text{datenum}(DT, F, P) \]
\[ N = \text{datenum}(Y, M, D) \]
\[ N = \text{datenum}(Y, M, D, H, MI, S) \]

Description

The `datenum` function converts date strings and date vectors (defined by `datevec`) into serial date numbers. Date numbers are serial days elapsed from some reference date. By default, the serial day 1 corresponds to 1-Jan-0000.

Date strings and date vectors can contain multiple dates in either a cell array of strings or an M-by-N vector, respectively. In either case, the resulting output is a column vector of date numbers.

\[ N = \text{datenum}(DT) \] converts the date string or date vector \( DT \) into a serial date number. Date strings with two-character years, e.g., 12-june-12, are assumed to lie within the 100-year period centered about the current year.

\[ N = \text{datenum}(DT, P) \] 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.

\[ N = \text{datenum}(DT, F) \] uses the specified date form \( F \) to interpret the date string \( DT \) during conversion to date number \( N \). The date form must be composed of date format symbols according to Table , Free-Form Date Format Specifiers, in the `datestr` function reference page.

\[ N = \text{datenum}(DT, F, P) \] uses the specified date form \( F \) to interpret the date string \( DT \) and pivot year \( P \) to interpret the year when expressed in two digits.

Note

If \( DT \) is a string, it must be in one of the date formats 0, 1, 2, 6, 13, 14, 15, 16, or 23 as defined by `datestr`.

\[ N = \text{datenum}(DT, P) \] 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.

\[ N = \text{datenum}(DT, F) \] uses the specified date form \( F \) to interpret the date string \( DT \) during conversion to date number \( N \). The date form must be composed of date format symbols according to Table , Free-Form Date Format Specifiers, in the `datestr` function reference page.

\[ N = \text{datenum}(DT, F, P) \] uses the specified date form \( F \) to interpret the date string \( DT \) and pivot year \( P \) to interpret the year when expressed in two digits.
N = datenum(Y, M, D) returns the serial date numbers for corresponding elements of the Y, M, and D (year, month, day) arrays. Y, M, and D must be arrays of the same size (or any can be a scalar). Values outside the normal range of each array are automatically carried to the next unit.

N = datenum(Y, M, D, H, MI, S) returns the serial date numbers for corresponding elements of the Y, M, D, H, MI, and S (year, month, day, hour, minute, and second) array values. Y, M, D, H, MI, and S must be arrays of the same size (or any 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.

Examples

Convert a date string to a serial date number:

   n = datenum('19-May-2001')

   n =
    730990

Specifying year, month, and day, convert a date to a serial date number:

   n = datenum(2001, 12, 19)

   n =
    731204

Convert a date vector to a serial date number:

   format bank
   n = datenum([2001 5 19 18 0 0])

   n =
    730990.75

Convert a date string to a serial date number using the default pivot year:

   n = datenum('12-june-12')

   n =
    735032
datenum

Convert the same date string to a serial date number using 1900 as the pivot year:

\[
n = \text{datenum('12\text{-}june\text{-}12', 1900)}
\]

\[
n = 698507
\]

Specify format 'dd.mm.yyyy' to be used in interpreting a nonstandard date string:

\[
n = \text{datenum('19.05.2000', 'dd.mm.yyyy')}
\]

\[
n = 730625.75
\]

See Also

datestr, datevec, date, clock, now, datetick
Purpose  
Date string format

Syntax  
\[
\text{str} = \text{datestr}(\text{DT})
\]
\[
\text{str} = \text{datestr}(\text{DT}, \text{dateform})
\]
\[
\text{str} = \text{datestr}(\text{DT}, \text{dateform}, P)
\]
\[
\text{str} = \text{datestr}(\ldots, 'local')
\]

Description  
\[
\text{str} = \text{datestr}(\text{DT}) \text{ converts a serial date number (defined by datenum) or date vector (defined by datevec) to a date string. You can also convert an array of } N \text{ serial date numbers or date vectors to an } N\text{-by-}M \text{ array of date strings.}
\]

Date strings with two-character years, e.g., \text{12\text{-}june\text{-}12}, are assumed to lie within the 100-year period centered about the current year.

\[
\text{str} = \text{datestr}(\text{DT}, \text{dateform}) \text{ converts a serial date number, date vector, or date string } \text{DT} \text{ to a date string having format } \text{dateform}. \text{The } \text{dateform} \text{ argument can be either a number or a string. See Table \text{, Dateform Format Numbers and Strings, on page 2\text{-}524, for valid } \text{dateform} \text{ values.}
\]

By default, the value of \text{dateform} is 1, 16, or 0, depending on whether \text{DT} contains a date, time, or both. If \text{DT} is a string, \text{dateform} must be one of 0, 1, 2, 6, 13, 14, 15, 16, or 23.

Table \text{, Free-Form Date Format Specifiers, on page 2\text{-}526, shows the symbols you can use to specify a free-form date format in the } \text{dateform} \text{ argument. These symbols control how MATLAB displays the returned string.}

\[
\text{str} = \text{datestr}(\text{DT}, \text{dateform}, P) \text{ uses the specified pivot year as the starting year of the 100-year range in which a two-character year resides. The default pivot year is the current year minus } 50 \text{ years.}
\]

\[
\text{str} = \text{datestr}(\ldots, 'local') \text{ returns the string in a localized format. The default is US English ('en\text{\_US}'). This argument must come last in the argument sequence.} 
\]
**Dateform Format Numbers and Strings**

<table>
<thead>
<tr>
<th>dateform (number)</th>
<th>dateform (string)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'dd-mmm-yyyy HH:MM:SS'</td>
<td>01-Mar-2000 15:45:17</td>
</tr>
<tr>
<td>1</td>
<td>'dd-mmm-yyyy'</td>
<td>01-Mar-2000</td>
</tr>
<tr>
<td>2</td>
<td>'mm/dd/yy'</td>
<td>03/01/00</td>
</tr>
<tr>
<td>3</td>
<td>'mmm'</td>
<td>Mar</td>
</tr>
<tr>
<td>4</td>
<td>'m'</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>'mm'</td>
<td>03</td>
</tr>
<tr>
<td>6</td>
<td>'mm/dd'</td>
<td>03/01</td>
</tr>
<tr>
<td>7</td>
<td>'dd'</td>
<td>01</td>
</tr>
<tr>
<td>8</td>
<td>'ddd'</td>
<td>Wed</td>
</tr>
<tr>
<td>9</td>
<td>'d'</td>
<td>W</td>
</tr>
<tr>
<td>10</td>
<td>'yyyy'</td>
<td>2000</td>
</tr>
<tr>
<td>11</td>
<td>'yy'</td>
<td>00</td>
</tr>
<tr>
<td>12</td>
<td>'mmmyy'</td>
<td>Mar00</td>
</tr>
<tr>
<td>13</td>
<td>'HH:MM:SS'</td>
<td>15:45:17</td>
</tr>
<tr>
<td>14</td>
<td>'HH:MM:SS PM'</td>
<td>3:45:17 PM</td>
</tr>
<tr>
<td>15</td>
<td>'HH:MM'</td>
<td>15:45</td>
</tr>
<tr>
<td>16</td>
<td>'HH:MM PM'</td>
<td>3:45 PM</td>
</tr>
<tr>
<td>17</td>
<td>'QQ-YY'</td>
<td>Q1-01</td>
</tr>
<tr>
<td>18</td>
<td>'QQ'</td>
<td>Q1</td>
</tr>
<tr>
<td>19</td>
<td>'dd/mm'</td>
<td>01/03</td>
</tr>
<tr>
<td>20</td>
<td>'dd/mm/yy'</td>
<td>01/03/00</td>
</tr>
</tbody>
</table>
### Dateform Format Numbers and Strings

<table>
<thead>
<tr>
<th>Dateform (number)</th>
<th>Dateform (string)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td><code>mmm.dd.yyyy HH:MM:SS</code></td>
<td>Mar.01,2000 15:45:17</td>
</tr>
<tr>
<td>22</td>
<td><code>mmm.dd.yyyy</code></td>
<td>Mar.01.2000</td>
</tr>
<tr>
<td>23</td>
<td><code>mm/dd/yyyy</code></td>
<td>03/01/2000</td>
</tr>
<tr>
<td>24</td>
<td><code>dd/mm/yyyy</code></td>
<td>01/03/2000</td>
</tr>
<tr>
<td>25</td>
<td><code>yy/mm/dd</code></td>
<td>00/03/01</td>
</tr>
<tr>
<td>26</td>
<td><code>yyyy/mm/dd</code></td>
<td>2000/03/01</td>
</tr>
<tr>
<td>27</td>
<td><code>QQ-YYYY</code></td>
<td>Q1-2001</td>
</tr>
<tr>
<td>28</td>
<td><code>mmm/yyyy</code></td>
<td>Mar2000</td>
</tr>
<tr>
<td>29 (ISO 8601)</td>
<td><code>yyyy-mm-dd</code></td>
<td>2000-03-01</td>
</tr>
<tr>
<td>30 (ISO 8601)</td>
<td><code>yyyyymmddTHHMMSS</code></td>
<td>20000301T154517</td>
</tr>
<tr>
<td>31</td>
<td><code>yyyy-mm-dd HH:MM:SS</code></td>
<td>2000-03-01 15:45:17</td>
</tr>
</tbody>
</table>

**Note**  
Dateform numbers 0, 1, 2, 6, 13, 14, 15, 16, and 23 produce a string suitable for input to `datenum` or `datevec`. Other date string formats do not work with these functions unless you specify a date form in the function call.

Time formats like `h:m:s`, `h:m:s.s`, `h:m pm`, ... can also be part of the input array DT. If you do not specify dateform, or if you specify dateform as -1, the date string format defaults to the following:

1. If DT contains date information only, e.g., 01-Mar-1995
2. If DT contains time information only, e.g., 03:45 PM
3. If DT is a date vector, or a string that contains both date and time information, e.g., 01-Mar-1995 03:45
The following table shows the string symbols to use in specifying a free-form format for the output date string. MATLAB interprets these symbols according to your computer’s language setting and the current MATLAB language setting.

**Free-Form Date Format Specifiers**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interpretation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy</td>
<td>Show year in full.</td>
<td>1990, 2002</td>
</tr>
<tr>
<td>YY</td>
<td>Show year in two digits.</td>
<td>90, 02</td>
</tr>
<tr>
<td>mmmm</td>
<td>Show month using full name.</td>
<td>March, December</td>
</tr>
<tr>
<td>mmm</td>
<td>Show month using first three letters.</td>
<td>Mar, Dec</td>
</tr>
<tr>
<td>mm</td>
<td>Show month in two digits.</td>
<td>03, 12</td>
</tr>
<tr>
<td>m</td>
<td>Show month using capitalized first letter.</td>
<td>M, D</td>
</tr>
<tr>
<td>dddd</td>
<td>Show day using full name.</td>
<td>Monday, Tuesday</td>
</tr>
<tr>
<td>ddd</td>
<td>Show day using first three letters.</td>
<td>Mon, Tue</td>
</tr>
<tr>
<td>dd</td>
<td>Show day in two digits.</td>
<td>05, 20</td>
</tr>
<tr>
<td>d</td>
<td>Show day using capitalized first letter.</td>
<td>M, T</td>
</tr>
<tr>
<td>HH</td>
<td>Show hour in two digits (no leading zeros when free-form specifier AM or PM is used (see last entry in this table)).</td>
<td>05, 5 AM</td>
</tr>
<tr>
<td>MM</td>
<td>Show minute in two digits.</td>
<td>12, 02</td>
</tr>
<tr>
<td>SS</td>
<td>Show second in two digits.</td>
<td>07, 59</td>
</tr>
<tr>
<td>AM or PM</td>
<td>Append AM or PM to date string (see note below).</td>
<td>3:45:02 PM</td>
</tr>
</tbody>
</table>

**Note** Free-form specifiers AM and PM from the table above are identical. They do not influence which characters are displayed following the time (AM versus
PM), but only whether or not they are displayed. MATLAB selects AM or PM based on the time entered.

**Examples**

Return the current date and time in a string using the default format, 0:

```matlab
datestr(now)
```

ans =
28-Jan-2003 13:41:27

Format the same showing only the date and in the mm/dd/yy format. Note that you can specify this format either by number or by string.

```matlab
datestr(now, 2) -or- datestr(now, 'mm/dd/yy')
```

ans =
01/28/03

Display the returned date string using your own format made up of symbols shown in the Free-Form Date Format Specifiers table above.

```matlab
datestr(now, 'dd.mm.yyyy')
```

ans =
28.01.2003

Convert a nonstandard date form into a standard MATLAB date form by first converting to a date number and then to a string:

```matlab
datestr(datum('24.01.2003', 'dd.mm.yyyy'), 2)
```

ans =
01/24/03

**See Also**
datenum, datevec, date, clock, now, datetick
**datetick**

**Purpose**
Label tick lines using dates

**Syntax**
- `datetick(tickaxis)`
- `datetick(tickaxis, dateform)`
- `datetick(...,'keeplimits')`
- `datetick(...,'keepticks')`
- `datetick(axes_handle,...)`

**Description**
`datetick(tickaxis)` labels the tick lines of an axis using dates, replacing the default numeric labels. `tickaxis` is the string 'x', 'y', or 'z'. The default is 'x'. `datetick` selects a label format based on the minimum and maximum limits of the specified axis.

`datetick(tickaxis, dateform)` formats the labels according to the integer `dateform` (see table). To produce correct results, the data for the specified axis must be serial date numbers (as produced by `datenum`).

<table>
<thead>
<tr>
<th>dateform (number)</th>
<th>dateform (string)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'dd-mm-yyyy HH:MM:SS'</td>
<td>01-Mar-2000 15:45:17</td>
</tr>
<tr>
<td>1</td>
<td>'dd-mm-yyyy'</td>
<td>01-Mar-2000</td>
</tr>
<tr>
<td>2</td>
<td>'mm/dd/yy'</td>
<td>03/01/00</td>
</tr>
<tr>
<td>3</td>
<td>'mmm'</td>
<td>Mar</td>
</tr>
<tr>
<td>4</td>
<td>'m'</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>'mm'</td>
<td>03</td>
</tr>
<tr>
<td>6</td>
<td>'mm/dd'</td>
<td>03/01</td>
</tr>
<tr>
<td>7</td>
<td>'dd'</td>
<td>01</td>
</tr>
<tr>
<td>8</td>
<td>'ddd'</td>
<td>Wed</td>
</tr>
<tr>
<td>9</td>
<td>'d'</td>
<td>W</td>
</tr>
<tr>
<td>10</td>
<td>'yyyy'</td>
<td>2000</td>
</tr>
<tr>
<td>11</td>
<td>'yy'</td>
<td>00</td>
</tr>
</tbody>
</table>
datetick

| 12 | 'mmmyy' | Mar00 |
| 13 | 'HH:MM:SS' | 15:45:17 |
| 14 | 'HH:MM SS PM' | 3:45:17 PM |
| 15 | 'HH:MM' | 15:45 |
| 16 | 'HH:MM PM' | 3:45 PM |
| 17 | 'QQ-YY' | Q1 01 |
| 18 | 'QQ' | Q1 |
| 19 | 'dd/mm' | 01/03 |
| 20 | 'dd/mm/yy' | 01/03/00 |
| 21 | 'mmm.dd.yyyy HH:MM:SS' | Mar.01,2000 15:45:17 |
| 22 | 'mmm.dd.yyyy' | Mar.01.2000 |
| 23 | 'mm/dd/yyyy' | 03/01/2000 |
| 24 | 'dd/mm/yyyy' | 01/03/2000 |
| 25 | 'yy/mm/dd' | 00/03/01 |
| 26 | 'yyyy/mm/dd' | 2000/03/01 |
| 27 | 'QQ-YYYY' | Q1 2001 |
| 28 | 'mmmyyyy' | Mar2000 |

datetick(...,'keeplimits') changes the tick labels to date-based labels while preserving the axis limits.

datetick(...,'keepticks') changes the tick labels to date-based labels without changing their locations.

You can use both keeplimits and keepticks in the same call to datetick.

datetick(axes_handle,...) uses the axes specified by the handle ax instead of the current axes.
**Remarks**

datetick calls datestr to convert date numbers to date strings.

To change the tick spacing and locations, set the appropriate axes property (i.e., XTick, YTick, or ZTick) before calling datetick.

**Example**

Consider graphing population data based on the 1990 U.S. census:

```matlab
t = (1900:10:1990)';  % Time interval
p = [75.995 91.972 105.711 123.203 131.669 ...
     150.697 179.323 203.212 226.505 249.633]';  % Population
plot(datenum(t,1,1),p)  % Convert years to date numbers and plot
grid on
datetick('x',11)  % Replace x-axis ticks with 2-digit year labels
```

**See Also**

The axes properties XTick, YTick, and ZTick

datenum, datestr

“Annotating Plots” for related functions
Purpose

Date components

\[ V = \text{datevec}(DT) \]
\[ V = \text{datevec}(DT, P) \]
\[ V = \text{datevec}(DT, F) \]
\[ V = \text{datevec}(DT, F, P) \]
\[ [Y, M, D, H, MI, S] = \text{datevec}(DT) \]

Description

\( V = \text{datevec}(DT) \) converts a serial date number (defined by \text{datenum}) or date string (defined by \text{datestr}) to a date vector \( V \) having elements \{year, month, day, hour, minute, second\}. The first five vector elements are integers. You can also convert an array of \( N \) serial date numbers or date strings to an \( N \)-by-6 array of date vectors.

Date strings with two-character years, e.g., \( 12\text{-june-12} \), are assumed to lie within the 100-year period centered about the current year.

\( V = \text{datevec}(DT, P) \) 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.

\( V = \text{datevec}(DT, F) \) uses the specified date form \( F \) to interpret the date string \( DT \) during conversion to date vector \( V \). The date form must be composed of date format symbols according to the Free-Form Date Format Specifiers table in the \text{datestr} function reference page.

\( V = \text{datevec}(DT, F, P) \) uses the specified date form \( F \) to interpret the date string \( DT \), and pivot year \( P \) to interpret the year when expressed in two digits.

\[ [Y, M, D, H, MI, S] = \text{datevec}(DT) \] returns the components of the date vector as individual variables.

When creating your own date vector, you need not make the components integers. Any components that lie outside their conventional ranges affect the next higher component (so that, for instance, the anomalous June 31 becomes July 1). A zeroth month, with zero days, is allowed.
Obtain a date vector using a string as input:

    datevec('12/24/1984 12:45')

    ans =
        1984    12    24    12    45       0

Obtain a date vector using a serial date number as input:

    t = datenum('12/24/1984 12:45')
    t =
        725000.53

    datevec(t)
    ans =
        1984    12    24    12    45       0

Assign elements of the returned date vector:

    [y, m, d, h, mi, s] = datevec('12/24/1984 12:45');

    sprintf('Date: %d/%d/%d   Time: %d:%d
', m, d, y, h, mi)

    ans =
        Date: 12/24/1984   Time: 12:45

Use free-form date format 'dd.mm.yyyy' to indicate how you want a nonstandard date string interpreted:

    datevec('19.05.2003', 'dd.mm.yyyy')

    ans =
        2003    19     5    12    45       0

See Also

datenum, datestr, date, clock, now, datetick
Purpose
Clear breakpoints

Graphical Interface
As an alternative to the dbclear function, there are various ways to clear breakpoints using the Editor/Debugger.

Syntax

```
dbclear all
dbclear in mfile
dbclear in mfile at lineno
dbclear in mfile at subfun
dbclear if caught error
dbclear if caught error identifier
dbclear if error
dbclear if error identifier
dbclear if warning
dbclear if warning identifier
dbclear if naninf
dbclear if infnan
```

Description

`dbclear all` removes all breakpoints in all M-files, as well as breakpoints set for errors, caught errors, caught error identifiers, warnings, warning identifiers, and naninf/infnan.

`dbclear in mfile` removes all breakpoints in `mfile`.

`dbclear in mfile at lineno` removes the breakpoint set at the line number `lineno` in `mfile`.

`dbclear in mfile at subfun` removes the breakpoint set at the subfunction `subfun` in `mfile`.

`dbclear if caught error` removes the breakpoints set using `dbstop if caught error` and `dbstop if caught error identifier` statements.

`dbclear if caught error identifier` removes the breakpoints set using the `dbstop if caught error identifier` statement for the specified identifier. It is an error to clear this setting on a specific identifier if `dbstop if caught error` or `dbstop if caught error all` is set.


**dbclear**

`dbclear if error` removing the breakpoints set using `dbstop if error` and `dbstop if error` identifier statements.

`dbclear if error` identifier removes the breakpoint set using `dbstop if error` identifier for the specified identifier. It is an error to clear this setting on a specific identifier if `dbstop if error` or `dbstop if error all` is set.

`dbclear if warning` removing the breakpoints set using the `dbstop if warning` and `dbstop if warning` identifier statements.

`dbclear if warning` identifier removes the breakpoint set using `dbstop if warning` identifier for the specified identifier. It is an error to clear this setting on a specific identifier if `dbstop if warning` or `dbstop if warning all` is set.

`dbclear if naninf` removing the breakpoint set by `dbstop if naninf`.

`dbclear if infnan` also removes the breakpoint set by `dbstop if naninf`.

**Remarks**

The `at`, and `in` keywords are optional.

**See Also**

`dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`, `partialpath`
Purpose  Resume execution

Graphical Interface  As an alternative to the dbcont function, you can select **Continue** from the **Debug** menu in the Editor/Debugger or click the **Continue** button in the Editor/Debugger toolbar.

Syntax  

```
dbcont
```

Description  dbcont resumes execution of an M-file from a breakpoint. Execution continues until another breakpoint is encountered, a pause condition is met, an error occurs, or MATLAB returns to the base workspace prompt.

See Also  dbclear, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup
dbdown

**Purpose**
Change local workspace context when in debug mode

**Graphical Interface**
As an alternative to the `dbdown` function, you can select a different workspace from the Stack field in the Editor/Debugger toolbar.

**Syntax**
dbdown

**Description**
dbdown changes the current workspace context to the workspace of the called M-file when a breakpoint is encountered. You must have issued the `dbup` function at least once before you issue this function. `dbdown` is the opposite of `dbup`.

Multiple `dbdown` functions change the workspace context to each successively executed M-file on the stack until the current workspace context is the current breakpoint. It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.

**See Also**
dbclear, dbcont, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup
**Purpose**
Numerically evaluate double integral

**Syntax**

\[
q = \text{dblquad}(\text{fun}, \text{xmin}, \text{xmax}, \text{ymin}, \text{ymax})
\]

\[
q = \text{dblquad}(\text{fun}, \text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}, \text{tol})
\]

\[
q = \text{dblquad}(\text{fun}, \text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}, \text{tol}, \text{method})
\]

**Description**

\[
q = \text{dblquad}(\text{fun}, \text{xmin}, \text{xmax}, \text{ymin}, \text{ymax})
\]
calls the \text{quad} function to evaluate the double integral \( \text{fun}(x,y) \) over the rectangle \( \text{xmin} \leq x \leq \text{xmax}, \text{ymin} \leq y \leq \text{ymax} \). \text{fun} is a function handle for either an M-file function or an anonymous function. \text{fun}(x,y) must accept a vector \( x \) and a scalar \( y \) and return a vector of values of the integrand.

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

\[
q = \text{dblquad}(\text{fun}, \text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}, \text{tol})
\]
uses a tolerance \( \text{tol} \) instead of the default, which is \( 1.0 \times 10^{-6} \).

\[
q = \text{dblquad}(\text{fun}, \text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}, \text{tol}, \text{method})
\]
uses the quadrature function specified as \( \text{method} \), instead of the default \text{quad}. Valid values for \( \text{method} \) are \( @\text{quadl} \) or the function handle of a user-defined quadrature method that has the same calling sequence as \text{quad} and \text{quadl}.

**Example**

Pass M-file function handle \( \text{integrnd} \) to \text{dblquad}:

\[
Q = \text{dblquad}(\text{integrnd}, \text{pi}, 2\times \text{pi}, 0, \text{pi})
\]

where the M-file \text{integrnd.m} is

\[
\text{function } z = \text{integrnd}(x, y) \\
z = y \times \sin(x) + x \times \cos(y);
\]

Pass anonymous function handle \( F \) to \text{dblquad}:

\[
F = @(x,y)y \times \sin(x) + x \times \cos(y);
\]

\[
Q = \text{dblquad}(F, \text{pi}, 2\times \text{pi}, 0, \text{pi})
\]

The \text{integrnd} function integrates \( y \times \sin(x) + x \times \cos(y) \) over the square \( \pi \leq x \leq 2\times \pi, 0 \leq y \leq \pi \). Note that the integrand can be evaluated with a vector \( x \) and a scalar \( y \).
Nonsquare regions can be handled by setting the integrand to zero outside of the region. For example, the volume of a hemisphere is

\[
\text{dblquad}(\theta(x,y)\sqrt{\max(1-(x.^2+y.^2),0)}), \ -1, 1, \ -1, 1
\]

or

\[
\text{dblquad}(\theta(x,y)\sqrt{1-(x.^2+y.^2)})\cdot(x.^2+y.^2\leq1), \ -1, 1, \ -1, 1
\]

**See Also**
quad, quadl, triplequad, @(function handle), anonymous functions
dbmex

Purpose
Enable MEX-file debugging

Syntax

```
dbmex on
dbmex off
dbmex stop
dbmex print
```

Description
dbmex on enables MEX-file debugging for UNIX platforms. It is not supported on the Sun Solaris platform. To use this option, first start MATLAB from within a debugger by typing `matlab -Ddebugger`, where debugger is the name of the debugger.

dbmex off disables MEX-file debugging.

dbmex stop returns to the debugger prompt.

dbmex print displays MEX-file debugging information.

Remarks

See Also
dbclear, dbcont, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup
**Purpose**  
Quit debug mode

**Graphical Interface**  
As an alternative to the `dbquit` function, you can select **Exit Debug Mode** from the **Debug** menu in the Editor/Debugger.

**Syntax**  
```
dbquit
```

**Description**  
`dbquit` immediately terminates the debugger and returns control to the base workspace prompt. The M-file being processed is *not* completed and no results are returned.

All breakpoints remain in effect.

**See Also**  
dbclear, dbcont, dbdown, dbstack, dbstatus, dbstep, dbstop, dbtype, dbup
**Purpose**
Display function call stack

**Graphical Interface**
As an alternative to the `dbstack` function, you can view the Stack field in the Editor/Debugger toolbar.

**Syntax**
```
dbstack
[ST,I] = dbstack
```

**Description**
`dbstack` displays the line numbers and M-file names of the function calls that led to the current breakpoint, listed in the order in which they were executed. The line number of the most recently executed function call (at which the current breakpoint occurred) is listed first, followed by its calling function, which is followed by its calling function, and so on, until the topmost M-file function is reached.

`dbstack(n)` omits from the display the first `n` frames. This is useful when issuing a `dbstack` from within, say, an error handler.

`dbstack('-completenames')` outputs the “complete name” (the absolute file name and the entire sequence of functions that nests the function in the stack frame) of each function in the stack.

Either none, one, or both of the `n` and `-completenames` may appear. If both appear, the order is irrelevant.

```
[ST,I] = dbstack returns the stack trace information in an m-by-1 structure ST with the fields
```

- **file** The file in which the function appears. This field will be the empty string if there is no file.
- **name** Function name within the file.
- **line** Function line number.

The current workspace index is returned in `I`.

If you step past the end of an M-file, then `dbstack` returns a negative line number value to identify that special case. For example, if the last line to be
dbstack

executed is line 15, then the dbstack line number is 15 before you execute that line and -15 afterwards.

Examples

dbstack

In /usr/local/matlab/toolbox/matlab/cond.m at line 13
   In test1.m at line 2
   In test.m at line 3

See Also
dbclear, dbcont, dbdown, dbquit, dbstatus, dbstep, dbstop, dbtype, dbup, mfilename
Purpose
List all breakpoints

Graphical Interface
Part of the information shown by `dbstatus` (namely, the breakpoint line numbers) is displayed graphically by the breakpoint icons when a file is viewed in the Editor/Debugger.

Syntax
```
dbstatus

dbstatus mfile

s = dbstatus(...)```

Description
`dbstatus` by itself lists all the breakpoints in effect including errors, caught errors, warnings, and `naninf`s.

`dbstatus mfile` displays a list of the line numbers for which breakpoints are set in the specified M-file.

`s = dbstatus(...)` returns the breakpoint information in an m-by-1 structure with the fields:

- `name` Function name.
- `line` Vector of breakpoint line numbers.
- `cond` Cell vector of breakpoint conditional expression strings corresponding to lines in the `line` field.
- `cond` Condition string (`'error'`, `'caught error'`, `'warning'`, or `'naninf'`).
- `identifier` When `cond` is one of `'error'`, `'caught error'`, or `'warning'`, a cell vector of MATLAB Message Identifier strings for which the particular `cond` state is set.

Use `dbstatus class/function`, `dbstatus private/function` or `dbstatus class/private/function` to determine the status for methods, private functions, or private methods (for a class named `class`). In all these forms you can further qualify the function name with a subfunction name as in `dbstatus function/subfunction`. 

2-543
dbstatus

See Also: dbclear, dbcont, dbdown, dbquit, dbstack, dbstep, dbstop, dbtype, dbup
Purpose

Execute one or more lines from current breakpoint

Graphical Interface

As an alternative to the dbstep function, you can select Step or Step In from the Debug menu in the Editor/Debugger, or click on the Step or Step In buttons of the Editor/Debugger toolbar.

Syntax

dbstep

dbstep nlines

dbstep in

dbstep out

Description

This function allows you to debug an M-file by following its execution from the current breakpoint. At a breakpoint, the dbstep function steps through execution of the current M-file one line at a time or at the rate specified by nlines.

$dbstep$, by itself, executes the next executable line of the current M-file. $dbstep$ steps over the current line, skipping any breakpoints set in functions called by that line.

$dbstep$ nlines executes the specified number of executable lines.

$dbstep$ in steps to the next executable line. If that line contains a call to another M-file function, execution will step to the first executable line of the called M-file function. If there is no call to an M-file on that line, $dbstep$ in is the same as $dbstep$.

$dbstep$ out runs the rest of the function and stops just after leaving the function.

For all forms, MATLAB also stops execution at any breakpoint it encounters.

See Also

dbclear, dbcont, dbdown, dbquit, dbstack, dbstatus, dbstop, dbtype, dbup
dbstop

Purpose  Set breakpoints

Graphical Interface  Some of the dbstop functionality can be accessed through the Debug menu or the toolbar buttons of the Editor/Debugger.

Syntax  

```
dbstop in mfile
```
```
dbstop in mfile at lineno
```
```
dbstop in mfile at lineno@n
```
```
dbstop in mfile at subfun
```
```
dbstop in mfile at lineno if expression
```
```
dbstop in mfile at lineno@n if expression
```
```
dbstop in mfile at subfun if expression
```
```
dbstop in mfile if expression
```
```
dbstop if error
```
```
dbstop if error identifier
```
```
dbstop if caught error
```
```
dbstop if caught error identifier
```
```
dbstop if warning
```
```
dbstop if warning identifier
```
```
dbstop if naninf
```
```
dbstop if infnan
```

Description  dbstop in mfile temporarily stops execution of mfile when you run it, at the first executable line, putting MATLAB in debug mode. mfile must be in a directory that is on the search path or in the current directory. If you have graphical debugging enabled, the MATLAB Debugger opens with a breakpoint at the first executable line of mfile. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use dbcont or dbstep to resume execution of mfile. Use dbquit to exit from the Debugger.

```
dbstop in mfile at lineno temporarily stops execution of mfile when you run it, just prior to execution of the line whose number is lineno, putting MATLAB in debug mode. mfile must be in a directory that is on the search path or in the current directory. If you have graphical debugging enabled, the MATLAB Debugger opens mfile with a breakpoint at line lineno. If that line`
is not executable, execution stops and the breakpoint is set at the next executable line following lineno. When execution stops, you can use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use dbcont or dbstep to resume execution of mfile. Use dbquit to exit from the Debugger.

**dbstop in mfile at lineno@** Stops just after any call to the first anonymous function in the specified line number in mfile.

**dbstop in mfile at lineno@n** Stops just after any call to the nth anonymous function in the specified line number in mfile.

**dbstop in mfile at subfun** temporarily stops execution of mfile when you run it, just prior to execution of the subfunction subfun, putting MATLAB in debug mode. mfile must be in a directory that is on the search path or in the current directory. If you have graphical debugging enabled, the MATLAB Debugger opens mfile with a breakpoint at the subfunction specified by subfun. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use dbcont or dbstep to resume execution of mfile. Use dbquit to exit from the Debugger.

**dbstop in mfile at lineno if expression** temporarily stops execution of mfile when you run it, just prior to execution of the line whose number is lineno, putting MATLAB in debug mode. Execution will stop only if expression evaluates to true. The expression, expression, is evaluated (as if by eval), in mfile's workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (true or false). mfile must be in a directory that is on the search path or in the current directory. If you have graphical debugging enabled, the MATLAB Debugger opens mfile with a breakpoint at line lineno. If that line is not executable, execution stops and the breakpoint is set at the next executable line following lineno. When execution stops, you can use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use dbcont or dbstep to resume execution of mfile. Use dbquit to exit from the Debugger.

**dbstop in mfile at lineno@ if expression** Stops just after any call to the first anonymous function in the specified line number in mfile if expression evaluates to true.
dbstop in mfile at lineno@n if expression Stops just after any call to the nth anonymous function in the specified line number in mfile if expression evaluates to true.

dbstop in mfile at subfun if expression temporarily stops execution of mfile when you run it, just prior to execution of the subfunction subfun, putting MATLAB in debug mode. Execution will stop only if expression evaluates to true. The expression, expression, is evaluated (as if by eval), in mfile's workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (true or false). mfile must be in a directory that is on the search path or in the current directory. If you have graphical debugging enabled, the MATLAB Debugger opens mfile with a breakpoint at the subfunction specified by subfun. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use dbcont or dbstep to resume execution of mfile. Use dbquit to exit from the Debugger.

dbstop in mfile if expression temporarily stops execution of mfile when you run it, at the first executable line, putting MATLAB in debug mode. Execution will stop only if expression evaluates to true. The expression, expression, is evaluated (as if by eval), in mfile's workspace when the breakpoint is encountered, and must evaluate to a scalar logical value (true or false). mfile must be in a directory that is on the search path or in the current directory. If you have graphical debugging enabled, the MATLAB Debugger opens with a breakpoint at the first executable line of mfile. You can then use the debugging utilities, review the workspace, or issue any valid MATLAB function. Use dbcont or dbstep to resume execution of mfile. Use dbquit to exit from the Debugger.

dbstop if error stops execution when any M-file you subsequently run produces a run-time error, putting MATLAB in debug mode, paused at the line that generated the error. The M-file must be in a directory that is on the search path or in the current directory. The errors that stop execution do not include run-time errors that are detected within a try...catch block. You cannot resume execution after an uncaught run-time error. Use dbquit to exit from the Debugger.

dbstop if error identifier stops execution when any M-file you subsequently run produces a run-time error whose message identifier is identifier, putting MATLAB in debug mode, paused at the line that
generated the error. The M-file must be in a directory that is on the search path or in the current directory. The errors that stop execution do not include run-time errors that are detected within a try...catch block. You cannot resume execution after an uncaught run-time error. Use dbquit to exit from the Debugger.

**dbstop if caught error** stops execution when any M-file you subsequently run produces a run-time error, putting MATLAB in debug mode, paused at the line that generated the error. The M-file must be in a directory that is on the search path or in the current directory. The errors that stop execution will only be those that are detected within a try...catch block. You cannot resume execution after an uncaught run-time error. Use dbquit to exit from the Debugger.

**dbstop if caught error identifier** stops execution when any M-file you subsequently run produces a run-time error whose message identifier is identifier, putting MATLAB in debug mode, paused at the line that generated the error. The M-file must be in a directory that is on the search path or in the current directory. The errors that stop execution will only be those that are detected within a try...catch block. You cannot resume execution after an uncaught run-time error. Use dbquit to exit from the Debugger.

**dbstop if warning** stops execution when any M-file you subsequently run produces a run-time warning, putting MATLAB in debug mode, paused at the line that generated the warning. The M-file must be in a directory that is on the search path or in the current directory. Use dbcont or dbstep to resume execution.

**dbstop if warning identifier** stops execution when any M-file you subsequently run produces a run-time warning whose message identifier is identifier, putting MATLAB in debug mode, paused at the line that generated the warning. The M-file must be in a directory that is on the search path or in the current directory. Use dbcont or dbstep to resume execution.

**dbstop if naninf or dbstop if infnan** stops execution when any M-file you subsequently run encounters an infinite value (Inf) or a value that is not a number (NaN), putting MATLAB in debug mode, paused at the line where Inf or NaN was encountered. For convenience, you can use either naninf or infnan—they perform in exactly the same manner. The M-file must be in a
dbstop

directory that is on the search path or in the current directory. Use dbcont or dbstep to resume execution. Use dbquit to exit from the Debugger.

Remarks

The at, and in keywords are optional.
Examples

The file buggy, used in these examples, consists of three lines.

    function z = buggy(x)
    n = length(x);
    z = (1:n)./x;

Stop at First Executable Line

The statements

    dbstop in buggy
    buggy(2:5)

stop execution at the first executable line in buggy

    n = length(x);

The function

    dbstep

advances to the next line, at which point you can examine the value of \( n \).

Stop if Error

Because buggy only works on vectors, it produces an error if the input \( x \) is a full
matrix. The statements

    dbstop if error
    buggy(magic(3))

produce

    ??? Error using ==> ./
    Matrix dimensions must agree.
    Error in ==> c:\buggy.m
    On line 3 ==> z = (1:n)./x;
    K>>

and put MATLAB in debug mode.
**dbstop**

*Stop if InfNaN*

In buggy, if any of the elements of the input `x` is zero, a division by zero occurs. The statements

```matlab
dbstop if naninf
buggy(0:2)
```

produce

```
Warning: Divide by zero.
> In c:\buggy.m at line 3
K>>
```

and put MATLAB in debug mode.

**See Also**

`break`, `dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbtype`, `dbup`, `keyboard`, `partialpath`, `return`
**Purpose**
List M-file with line numbers

**Graphical Interface**
As an alternative to the `dbtype` function, you can see an M-file with line numbers by opening it in the Editor/Debugger.

**Syntax**
- `dbtype mfile`
- `dbtype mfile start:end`

**Description**
The `dbtype` command is used to list an M-file function with line numbers to aid the user in setting breakpoints.

- `dbtype mfile` displays the contents of the specified M-file function with line numbers preceding each line. `mfile` must be full path name of an M-file function or a MATLAB path relative partial path name.

- `dbtype mfile start:end` displays the portion of the file specified by a range of line numbers from `start` to `end`.

You cannot use `dbtype` for built-in functions.

**Examples**
To see only the input and output arguments for a function, that is, the first line of the M-file, type

```
dbtype mfile 1
```

For example,

```
dbtype fileparts 1
```
returns

```
1     function [path, fname, extension,version] = fileparts(name)
```

**See Also**
- `dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbup`, `partialpath`
**dbup**

**Purpose**  
Change local workspace context

**Graphical Interface**  
As an alternative to the dbup function, you can select a different workspace from the **Stack** field in the toolbar of the Editor/Debugger.

**Syntax**  
```
dbup
```

**Description**  
This function allows you to examine the calling M-file to determine what led to the arguments’ being passed to the called function.

`dbup` changes the current workspace context, while the user is in the debug mode, to the workspace of the calling M-file.

Multiple `dbup` functions change the workspace context to each previous calling M-file on the stack until the base workspace context is reached. (It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.)

**See Also**  
dbclear, dbcont, dbdown, dbquit, dbstack, dbstatus, dbstep, dbstop, dbtype
Purpose
Solve delay differential equations (DDEs) with constant delays

Syntax
sol = dde23(ddefun, lags, history, tspan)
sol = dde23(ddefun, lags, history, tspan, options)

Arguments
ddefun Function that evaluates the right side of the differential
equations \( y'(t) = f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k)) \). The function
must have the form
\[
dydt = ddefun(t, y, Z)
\]
where \( t \) corresponds to the current \( t \), \( y \) is a column vector that
approximates \( y(t) \), and \( Z(:, j) \) approximates \( y(t - \tau_j) \) for
delay \( \tau_j = lags(j) \). The output is a column vector
 corresponding to \( f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k)) \).
lags Vector of constant, positive delays \( \tau_1, ..., \tau_k \).
history Specify history in one of three ways:
• A function of \( t \) such that \( y = history(t) \) returns the
  solution \( y(t) \) for \( t \leq t_0 \) as a column vector
• A constant column vector, if \( y(t) \) is constant
• The solution \( sol \) from a previous integration, if this call
  continues that integration
tspan Interval of integration as a vector \([t_0, tf]\) with \( t_0 < tf \).
options Optional integration argument. A structure you create using
  the ddeset function. See ddeset for details.
p1,p2,... Optional parameters that dde23 passes to ddefun, if it is a
  function, and any functions you specify in options.

Description
sol = dde23(ddefun, lags, history, tspan) integrates the system of DDEs
\[
y'(t) = f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))
\]
on the interval \([t_0, tf]\), where \( \tau_1, ..., \tau_k \) are constant, positive delays and
\( t_0 < tf \).
Parameterizing Functions Called by Function Functions, in the online MATLAB documentation, explains how to provide additional parameters to the function ddefun, if necessary.

dde23 returns the solution as a structure sol. Use the auxiliary function deval and the output sol to evaluate the solution at specific points tint in the interval tspan = [t0,tf].

\[
yint = \text{deval}(\text{sol}, \text{tint})
\]

The structure sol returned by dde23 has the following fields.

- sol.x: Mesh selected by dde23
- sol.y: Approximation to \( y(x) \) at the mesh points in sol.x.
- sol.yp: Approximation to \( y'(x) \) at the mesh points in sol.x
- sol.solver: Solver name, 'dde23'

\[
sol = \text{dde23}(\text{ddefun}, \text{lags}, \text{history}, \text{tspan}, \text{options})
\]
solves as above with default integration properties replaced by values in options, an argument created with ddeset. See ddeset and “Initial Value Problems for DDEs” in the MATLAB documentation for details.

Commonly used options are scalar relative error tolerance ‘RelTol’ (1e-3 by default) and vector of absolute error tolerances ‘AbsTol’ (all components are 1e-6 by default).

Use the ‘Jumps’ option to solve problems with discontinuities in the history or solution. Set this option to a vector that contains the locations of discontinuities in the solution prior to t0 (the history) or in coefficients of the equations at known values of \( t \) after t0.

Use the ‘Events’ option to specify a function that dde23 calls to find where functions \( g(t,y(t),y(t-\tau_1),...,y(t-\tau_k)) \) vanish. This function must be of the form

\[
[\text{value,isterminal,direction}] = \text{events}(t,y,Z)
\]

and contain an event function for each event to be tested. For the kth event function in events:

- value(k) is the value of the kth event function.
• `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and 0 otherwise.
• `direction(k) = 0` if you want dde23 to compute all zeros of this event function, +1 if only zeros where the event function increases, and -1 if only zeros where the event function decreases.

If you specify the 'Events' option and events are detected, the output structure `sol` also includes fields:

- `sol.xe` Row vector of locations of all events, i.e., times when an event function vanished
- `sol.ye` Matrix whose columns are the solution values corresponding to times in `sol.xe`
- `sol.ie` Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

**Examples**

This example solves a DDE on the interval [0, 5] with lags 1 and 0.2. The function `ddex1de` computes the delay differential equations, and `ddex1hist` computes the history for \( t \leq 0 \).

```
sol = dde23(@ddex1de,[1, 0.2],@ddex1hist,[0, 5]);
```

This code evaluates the solution at 100 equally spaced points in the interval [0,5], then plots the result.
```
tint = linspace(0,5);
yint = deval(sol,tint);
plot(tint,yint);
```

`ddex1` shows how you can code this problem using subfunctions. For more examples see `ddex2`.

**Note** The demo `ddex1` contains the complete code for this example. To see the code in an editor, click the example name, or type `edit ddex1` at the command line. To run the example type `ddex1` at the command line.
**dde23**

**Algorithm**

dde23 tracks discontinuities and integrates with the explicit Runge-Kutta (2,3) pair and interpolant of ode23. It uses iteration to take steps longer than the lags.

**See Also**

ddeget, ddeset, deval, @(function_handle)

**References**

Purpose

Extract properties from options structure created with ddeset

Syntax

val = ddeget(options,'name')
val = ddeget(options,'name',default)

Description

val = ddeget(options,'name') extracts the value of the named property from the structure options, returning an empty matrix if the property value is not specified in options. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. [ ] is a valid options argument.

val = ddeget(options,'name',default) extracts the named property as above, but returns val = default if the named property is not specified in options. For example,

val = ddeget(opts,'RelTol',1e-4);

returns val = 1e-4 if the RelTol is not specified in opts.

See Also

dde23, ddeset
**ddeset**

**Purpose**  
Create/alter delay differential equations (DDE) options structure

**Syntax**  
options = ddeset('name1',value1,'name2',value2,...)  
options = ddeset(oldopts,'name1',value1,...)  
options = ddeset(oldopts,newopts)

ddeset

**Description**  
options = ddeset('name1',value1,'name2',value2,...) creates an integrator options structure options in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names.

options = ddeset(oldopts,'name1',value1,...) alters an existing options structure oldopts.

options = ddeset(oldopts,newopts) combines an existing options structure oldopts with a new options structure newopts. Any new properties overwrite corresponding old properties.

ddeset with no input arguments displays all property names and their possible values.

**DDE Properties**  
These properties are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelTol</td>
<td>Positive scalar {1e-3}</td>
<td>Relative error tolerance that applies to all components of the solution vector. The estimated error in each integration step satisfies $</td>
</tr>
<tr>
<td>AbsTol</td>
<td>Positive scalar or vector {1e-6}</td>
<td>Absolute error tolerance that applies to all components of the solution vector. Elements of a vector of tolerances apply to corresponding components of the solution vector.</td>
</tr>
</tbody>
</table>
Control error relative to norm of solution. Set this property on to request that dde23 control the error in each integration step with \( \|e\| \leq \max(\text{RelTol} \cdot \|y\|, \text{AbsTol}) \). By default dde23 uses a more stringent component-wise error control.

Display computational cost statistics.

The solver uses the specified function to locate where functions of \( t, y, z \) vanish. See dde23 for details.

Upper bound on the magnitude of the step size. The default is one-tenth of the \( tspan \) interval.

Suggested initial step size. The solver tries this first. By default the solver determines an initial step size automatically.

Installable output function. This output function is called by the solver after each time step. When a solver is called with no output arguments, \( \text{OutputFcn} \) defaults to the function \( \text{odeplot} \). Otherwise, \( \text{OutputFcn} \) defaults to \( [] \).

To create or modify an output function, see ODE Solver Output Properties in the “Differential Equations” section of the MATLAB documentation.

Output selection indices. Specifies the components of the solution vector that dde23 passes to the \( \text{OutputFcn} \). The default is all components.
Jumps

Vector

Location of discontinuities in solution. Points $t$ where the history or solution may have a jump discontinuity in a low-order derivative. See dde23 for details.

InitialY

Vector

Initial value of solution. By default the initial value of the solution is the value returned by history at the initial point. A different initial value can be supplied as the value of the InitialY property.

See Also

dde23, ddeget, @(function_handle)
**Purpose**
Deal inputs to outputs

**Syntax**

\[
[Y_1,Y_2,Y_3,...] = \text{deal}(X)
\]

\[
[Y_1,Y_2,Y_3,...] = \text{deal}(X_1,X_2,X_3,...)
\]

**Description**

\[
[Y_1,Y_2,Y_3,...] = \text{deal}(X)
\]
copies the single input to all the requested outputs. It is the same as
\[
Y_1 = X, \quad Y_2 = X, \quad Y_3 = X, \ldots
\]

\[
[Y_1,Y_2,Y_3,...] = \text{deal}(X_1,X_2,X_3,...)
\]
is the same as
\[
Y_1 = X_1, \quad Y_2 = X_2, \quad Y_3 = X_3, \ldots
\]

**Remarks**
deal is most useful when used with cell arrays and structures via comma-separated list expansion. Here are some useful constructions:

\[
[S.\text{field}] = \text{deal}(X)
\]
sets all the fields with the name field in the structure array S to the value X. If S doesn't exist, use \[
[S(1:m).\text{field}] = \text{deal}(X).
\]

\[
[X{:}] = \text{deal}(A.\text{field})
\]
copies the values of the field with name field to the cell array X. If X doesn't exist, use \[
[X{1:m}] = \text{deal}(A.\text{field}).
\]

\[
[Y_1,Y_2,Y_3,...] = \text{deal}(X{:})
\]
copies the contents of the cell array X to the separate variables Y1,Y2,Y3,...

\[
[Y_1,Y_2,Y_3,...] = \text{deal}(S.\text{field})
\]
copies the contents of the fields with the name field to separate variables Y1,Y2,Y3,...

**Examples**
Use deal to copy the contents of a 4-element cell array into four separate output variables.

\[
C = \{\text{rand}(3) \quad \text{ones}(3,1) \quad \text{eye}(3) \quad \text{zeros}(3,1)\};
\]

\[
[a,b,c,d] = \text{deal}(C{:})
\]

\[
a =
\begin{bmatrix}
0.9501 & 0.4860 & 0.4565 \\
0.2311 & 0.8913 & 0.0185 \\
0.6068 & 0.7621 & 0.8214
\end{bmatrix}
\]

\[
b =
\]
1
1
1
c =

1  0  0
0  1  0
0  0  1
d =

0
0
0

Use deal to obtain the contents of all the name fields in a structure array:

A.name = 'Pat'; A.number = 176554;
A(2).name = 'Tony'; A(2).number = 901325;
[name1,name2] = deal(A(:).name)

name1 =
Pat
name2 =
Tony

Note  In many instances, you can access the data in cell arrays and structure
fields without using the deal function.

These two commands perform the same operation as those used in the previous
two examples, except that these commands do not require deal.

[a,b,c,d] = C{:}
[name1,name2] = A(:).name
See Also

cell, iscell, celldisp, struct, isstruct, fieldnames, isfield, orderfields, rmfield, cell2struct, struct2cell
**Purpose**  Strip trailing blanks from the end of a string

**Syntax**  

```matlab
str = deblank(str)
c = deblank(c)
```

**Description**  

`str = deblank(str)` removes the trailing blanks from the end of a character string `str`.

`c = deblank(c),` when `c` is a cell array of strings, applies `deblank` to each element of `c`.

The `deblank` function is useful for cleaning up the rows of a character array.

**Examples**

```matlab
A{1,1} = 'MATLAB    ';
A{1,2} = 'SIMULINK    ';
A{2,1} = 'Toolboxes    ';
A{2,2} = 'The MathWorks    ';
A =
   'MATLAB'    'SIMULINK'
   'Toolboxes'    'The MathWorks'

debank(A)
ans =
   'MATLAB'    'SIMULINK'
   'Toolboxes'    'The MathWorks'
```
### Purpose
M-file debugging functions

### Graphical Interface
As an alternative to the debugging functions, you can use debugging features in the **Debug** menu and toolbar buttons of the Editor/Debugger.

### Description
Use debugging functions (listed in the See Also section) to help you identify problems in your M-files.

Set breakpoints using `dbstop`.

When a breakpoint is hit during execution, MATLAB goes into debug mode, the debugger window becomes active, and the prompt changes to a `K>>`. Any MATLAB command is allowed at the prompt.

To resume execution, use `dbcont` or `dbstep`. To exit from the debugger use `dbquit`.

### See Also
- `dbclear`, `dbcont`, `dbdown`, `dbquit`, `dbstack`, `dbstatus`, `dbstep`, `dbstop`, `dbtype`, `dbup`

Debugging M-Files in the MATLAB documentation details the Editor/Debugger as well as the use of debugging functions.
**Purpose**  
Decimal number to base conversion

**Syntax**  
str = dec2base(d,base)  
str = dec2base(d,base,n)

**Description**  
str = dec2base(d,base) converts the nonnegative integer d to the specified base. d must be a nonnegative integer smaller than $2^{52}$, and base must be an integer between 2 and 36. The returned argument str is a string.

str = dec2base(d,base,n) produces a representation with at least n digits.

**Examples**  
The expression `dec2base(23,2)` converts $23_{10}$ to base 2, returning the string '10111'.

**See Also**  
base2dec
Purpose
Decimal to binary number conversion

Syntax
str = dec2bin(d)
str = dec2bin(d,n)

Description
str = dec2bin(d) returns the binary representation of d as a string. d must be a nonnegative integer smaller than 2^52.

str = dec2bin(d,n) produces a binary representation with at least n bits.

Examples
ans =
     10111

See Also
bin2dec, dec2hex
**dec2hex**

**Purpose**
Decimal to hexadecimal number conversion

**Syntax**
str = dec2hex(d)
str = dec2hex(d,n)

**Description**
str = dec2hex(d) converts the decimal integer d to its hexadecimal representation stored in a MATLAB string. d must be a nonnegative integer smaller than 2^52.

str = dec2hex(d,n) produces a hexadecimal representation with at least n digits.

**Examples**
To convert decimal 1023 to hexadecimal,

dec2hex(1023)

ans =
   3FF

**See Also**
dec2bin, format, hex2dec, hex2num
Purpose
Compute consistent initial conditions for ode15i.

Syntax
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options)
[y0mod,yp0mod] = 
    decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options,p1,p2...) 
[y0mod,yp0mod,resnrm] = decic(...)

Description
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0) uses the inputs y0 and yp0 as initial guesses for an iteration to find output values that satisfy the requirement \( f(t_0,y_{0mod},y_{p0mod}) = 0 \), i.e., y0mod and yp0mod are consistent initial conditions. The function decic changes as few components of the guesses as possible. You can specify that decic holds certain components fixed by setting \( \text{fixed}_y0(i) = 1 \) if no change is permitted in the guess for \( y0(i) \) and 0 otherwise. decic interprets \( \text{fixed}_y0 = [] \) as allowing changes in all entries. \( \text{fixed}_yp0 \) is handled similarly.

You cannot fix more than length(y0) components. Depending on the problem, it may not be possible to fix this many. It also may not be possible to fix certain components of y0 or yp0. It is recommended that you fix no more components than necessary.

[y0mod,yp0mod] = 
    decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options) computes as above with default tolerances for consistent initial conditions, AbsTol and RelTol, replaced by the values in options, a structure you create with the odeset function.

[y0mod,yp0mod] = 
    decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options,p1,p2...) passes the additional parameters \( p1,p2... \) to the ODE function as \( \text{odefun}(t,y,yp,p1,p2...) \), and to all functions specified in options. Use options = [] as a place holder if no options are set.

[y0mod,yp0mod,resnrm] = 
    decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0...) returns the norm of \( \text{odefun}(t0,y_{0mod},y_{p0mod}) \) as resnrm. If the norm seems unduly large, use options to decrease RelTol (1e-3 by default).
These demos provide examples of the use of decic in solving implicit ODEs: ihb1dae, iburgersode.

ode15i, odeget, odeset
Purpose
Deconvolution and polynomial division

Syntax
\[ [q,r] = \text{deconv}(v,u) \]

Description
\[ [q,r] = \text{deconv}(v,u) \] deconvolves vector \( u \) out of vector \( v \), using long division. The quotient is returned in vector \( q \) and the remainder in vector \( r \) such that \( v = \text{conv}(u,q) + r \).

If \( u \) and \( v \) are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division. The result of dividing \( v \) by \( u \) is quotient \( q \) and remainder \( r \).

Examples
If
\[
\begin{align*}
  u & = [1 \ 2 \ 3 \ 4] \\
  v & = [10 \ 20 \ 30]
\end{align*}
\]
the convolution is
\[
\begin{align*}
  c & = \text{conv}(u,v) \\
  c & = \\
  & 10 \ 40 \ 100 \ 160 \ 170 \ 120
\end{align*}
\]

Use deconvolution to recover \( u \):
\[
\begin{align*}
  [q,r] & = \text{deconv}(c,u) \\
  q & = \\
  & 10 \ 20 \ 30 \\
  r & = \\
  & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0
\end{align*}
\]
This gives a quotient equal to \( v \) and a zero remainder.

Algorithm
deconv uses the filter primitive.

See Also
conv, residue
Purpose
Discrete Laplacian

Syntax
\[ L = \text{del2}(U) \]  
\[ L = \text{del2}(U,h) \]  
\[ L = \text{del2}(U,hx,hy) \]  
\[ L = \text{del2}(U,hx,hy,hz,\ldots) \]

Definition
If the matrix \( U \) is regarded as a function \( u(x,y) \) evaluated at the point on a square grid, then \( 4 \cdot \text{del2}(U) \) is a finite difference approximation of Laplace’s differential operator applied to \( u \), that is:

\[
L = \nabla^2 u = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right)
\]

where:

\[
l_{ij} = \frac{1}{4} (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) - u_{i,j}
\]

in the interior. On the edges, the same formula is applied to a cubic extrapolation.

For functions of more variables \( u(x,y,z,\ldots) \), \( \text{del2}(U) \) is an approximation,

\[
l = \nabla^2 u = \frac{1}{2N} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} + \frac{d^2 u}{dz^2} + \ldots \right)
\]

where \( N \) is the number of variables in \( u \).

Description
\[ L = \text{del2}(U) \] where \( U \) is a rectangular array is a discrete approximation of

\[
l = \nabla^2 u = \frac{1}{4} \left( \frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} \right)
\]

The matrix \( L \) is the same size as \( U \) with each element equal to the difference between an element of \( U \) and the average of its four neighbors.
- \( L = \text{del2}(U) \) when \( U \) is an multidimensional array, returns an approximation of

\[
\frac{\nabla^2 u}{2N}
\]

where \( N \) is \text{ndims}(u).

\( L = \text{del2}(U,h) \) where \( H \) is a scalar uses \( H \) as the spacing between points in each direction (\( h=1 \) by default).

\( L = \text{del2}(U,hx,hy) \) when \( U \) is a rectangular array, uses the spacing specified by \( hx \) and \( hy \). If \( hx \) is a scalar, it gives the spacing between points in the x-direction. If \( hx \) is a vector, it must be of length \( \text{size}(u,2) \) and specifies the x-coordinates of the points. Similarly, if \( hy \) is a scalar, it gives the spacing between points in the y-direction. If \( hy \) is a vector, it must be of length \( \text{size}(u,1) \) and specifies the y-coordinates of the points.

\( L = \text{del2}(U,hx,hy,hz,...) \) where \( U \) is multidimensional uses the spacing given by \( hx, hy, hz, ... \)

**Examples**

The function

\[
u(x, y) = x^2 + y^2
\]

has

\[
\nabla^2 u = 4
\]

For this function, \( 4*\text{del2}(U) \) is also 4.

\[
[x, y] = \text{meshgrid}(-4:4,-3:3);
U = x.*x+y.*y
U =
\begin{array}{cccccccc}
25 & 18 & 13 & 10 & 9 & 10 & 13 & 18 & 25 \\
20 & 13 & 8 & 5 & 4 & 5 & 8 & 13 & 20 \\
17 & 10 & 5 & 2 & 1 & 2 & 5 & 10 & 17 \\
16 & 9 & 4 & 1 & 0 & 1 & 4 & 9 & 16 \\
17 & 10 & 5 & 2 & 1 & 2 & 5 & 10 & 17 \\
20 & 13 & 8 & 5 & 4 & 5 & 8 & 13 & 20 \\
25 & 18 & 13 & 10 & 9 & 10 & 13 & 18 & 25
\end{array}
\]
See Also  diff, gradient
Purpose

Delaunay triangulation

Syntax

TRI = delaunay(x,y)
TRI = delaunay(x,y,options)

Definition

Given a set of data points, the Delaunay triangulation is a set of lines connecting each point to its natural neighbors. The Delaunay triangulation is related to the Voronoi diagram— the circle circumscribed about a Delaunay triangle has its center at the vertex of a Voronoi polygon.

Description

TRI = delaunay(x,y) for the data points defined by vectors x and y, returns a set of triangles such that no data points are contained in any triangle's circumscribed circle. Each row of the m-by-3 matrix TRI defines one such triangle and contains indices into x and y. If the original data points are collinear or x is empty, the triangles cannot be computed and delaunay returns an empty matrix.

delaunay uses Qhull.

TRI = delaunay(x,y,options) specifies a cell array of strings options to be used in Qhull via delaunayn. The default options are {'Qt','Qbb','Qc'}. If options is [], the default options are used. If options is {''}, no options are used, not even the default. For more information on Qhull and its options, see http://www.qhull.org.

Remarks

The Delaunay triangulation is used by: griddata (to interpolate scattered data), voronoi (to compute the voronoi diagram), and is useful by itself to create a triangular grid for scattered data points.
The functions dsearch and tsearch search the triangulation to find nearest neighbor points or enclosing triangles, respectively.

**Visualization**

Use one of these functions to plot the output of `delaunay`:

- **triplot** Displays the triangles defined in the \( m \)-by-3 matrix \( TRI \). See Example 1.

- **trisurf** Displays each triangle defined in the \( m \)-by-3 matrix \( TRI \) as a surface in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example

  \[
  \text{trisurf}(TRI,x,y,zeros(size(x)))
  \]

  See Example 2.

- **trimesh** Displays each triangle defined in the \( m \)-by-3 matrix \( TRI \) as a mesh in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example,

  \[
  \text{trimesh}(TRI,x,y,zeros(size(x)))
  \]

  produces almost the same result as triplot, except in 3-D space. See Example 2.

**Examples**

**Example 1.** Plot the Delaunay triangulation for 10 randomly generated points.

```matlab
rand('state',0);
x = rand(1,10);
y = rand(1,10);
TRI = delaunay(x,y);
subplot(1,2,1),...
triplot(TRI,x,y) 
axis([0 1 0 1]);
hold on;
plot(x,y,'or');
hold off
```

Compare the Voronoi diagram of the same points:

```matlab
[vx, vy] = voronoi(x,y,TRI);
subplot(1,2,2),...
plot(x,y,'r+',vx,vy,'b-'),...
```
Example 2. Create a 2-D grid then use `trisurf` to plot its Delaunay triangulation in 3-D space by using 0s for the third dimension.

```
[x,y] = meshgrid(1:15,1:15);
tri = delaunay(x,y);
trisurf(tri,x,y,zeros(size(x)))
```
Next, generate peaks data as a 15-by-15 matrix, and use that data with the Delaunay triangulation to produce a surface in 3-D space.

```matlab
z = peaks(15);
trisurf(tri,x,y,z)
```
You can use the same data with `trimesh` to produce a mesh in 3-D space.

```
trimesh(tri,x,y,z)
```

**Algorithm**

delaunay is based on Qhull. For information about Qhull, see [http://www.qhull.org/](http://www.qhull.org/). For copyright information, see [http://www.qhull.org/COPYING.txt](http://www.qhull.org/COPYING.txt).

**See Also**
delaunay3, delaunayn, dsearch, griddata, plot, triplot, trimesh, trisurf, tsearch, voronoi

**References**


**Purpose**  
3-dimensional Delaunay tessellation

**Syntax**  
\[ \text{T} = \text{delaunay3}(x,y,z) \]  
\[ \text{T} = \text{delaunay3}(x,y,z,\text{options}) \]

**Description**  
\( \text{T} = \text{delaunay3}(x,y,z) \) returns an array \( \text{T} \), each row of which contains the indices of the points in \((x,y,z)\) that make up a tetrahedron in the tessellation of \((x,y,z)\). \( \text{T} \) is a \( \text{numtes} \)-by-4 array where \( \text{numtes} \) is the number of facets in the tessellation. \( x \), \( y \), and \( z \) are vectors of equal length. If the original data points are collinear or \( x \), \( y \), and \( z \) define an insufficient number of points, the triangles cannot be computed and \text{delaunay3} returns an empty matrix.  
\text{delaunay3} uses Qhull.  
\[ \text{T} = \text{delaunay3}(x,y,z,\text{options}) \] specifies a cell array of strings \text{options} to be used in Qhull via \text{delaunay3}. The default options are \{'Qt', 'Qbb', 'Qc'\}.  
If \text{options} is [], the default options are used. If \text{options} is '{}', no options are used, not even the default. For more information on Qhull and its options, see [http://www.qhull.org](http://www.qhull.org).

**Visualization**  
Use \text{tetramesh} to plot \text{delaunay3} output. \text{tetramesh} displays the tetrahedrons defined in \( \text{T} \) as mesh. \text{tetramesh} uses the default transparency parameter value 'FaceAlpha' = 0.9.

**Example**  
This example generates a 3-dimensional Delaunay tessellation, then uses \text{tetramesh} to plot the tetrahedrons that form the corresponding simplex. \text{camorbit} rotates the camera position to provide a meaningful view of the figure.  
\[ d = [-1 1]; \]  
\[ [x,y,z] = \text{meshgrid}(d,d,d); \quad \% \text{A cube} \]  
\[ x = [x(:);0]; \]  
\[ y = [y(:);0]; \]  
\[ z = [z(:);0]; \]  
\% [x,y,z] are corners of a cube plus the center.  
\[ \text{Tes} = \text{delaunay3}(x,y,z) \]  
\[ \text{Tes} = \]
Algorithm \texttt{delaunay3} is based on Qhull [2]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

\begin{verbatim}
X = [x(:) y(:) z(:)];
tetramesh(Tes,X);camorbit(20,0)
\end{verbatim}
delaunay3

See Also
delaunay, delaunayn

Reference

Purpose

N-dimensional Delaunay tessellation

Syntax

\[
T = delaunayn(X) \\
T = delaunayn(X, \text{options})
\]

Description

\(T = \text{delaunayn}(X)\) computes a set of simplices such that no data points of \(X\) are contained in any circumspheres of the simplices. The set of simplices forms the Delaunay tessellation. \(X\) is an \(m\) by \(n\) array representing \(m\) points in \(n\)-dimensional space. \(T\) is a \(numt\) by \((n+1)\) array where each row contains the indices into \(X\) of the vertices of the corresponding simplex.

delaunayn uses Qhull.

\(T = \text{delaunayn}(X, \text{options})\) specifies a cell array of strings \(\text{options}\) to be used as options in Qhull. The default options are:

- \{'Qt','Qbb','Qc'\} for 2- and 3-dimensional input
- \{'Qt','Qbb','Qc','Qx'\} for 4 and higher-dimensional input

If \(\text{options}\) is [], the default options used. If \(\text{options}\) is {''}, no options are used, not even the default. For more information on Qhull and its options, see http://www.qhull.org.

Visualization

Plotting the output of \(\text{delaunayn}\) depends of the value of \(n\):

- For \(n = 2\), use triplot, trisurf, or trimesh as you would for delaunay.
- For \(n = 3\), use tetramesh as you would for delaunay3.
  For more control over the color of the facets, use patch to plot the output. For an example, see “Tessellation and Interpolation of Scattered Data in Higher Dimensions” in the MATLAB documentation.
- You cannot plot delaunayn output for \(n > 3\).

Example

This example generates an \(n\)-dimensional Delaunay tessellation, where \(n = 3\).

\[
d = [-1 1]; \\
[x,y,z] = \text{meshgrid}(d,d,d); \quad \% \text{A cube} \\
x = [x(:);0]; \\
y = [y(:);0]; \\
z = [z(:);0]; \\
\% [x,y,z] are corners of a cube plus the center.
\]
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

You can use tetramesh to visualize the tetrahedrons that form the corresponding simplex. camorbit rotates the camera position to provide a meaningful view of the figure.

tetramesh(Tes,X);camorbit(20,0)
Algorithm  

delaunayn is based on Qhull [2]. For information about Qhull, see http://www.qhull.org/. For copyright information, see http://www.qhull.org/COPYING.txt.

See Also  

convhulln, delaunayn, delaunay3, tetramesh, voronoin

Reference  


Purpose
Delete files or graphics objects

Graphical Interface
As an alternative to the delete function, you can delete files using the Current Directory browser.

Syntax
delete(filename)
delete(h)
delete('filename')

Description
delete(filename) deletes the named file from the disk. The filename may include an absolute pathname or a pathname relative to the current directory. The filename may also include wildcards, (*).

delete(h) deletes the graphics object with handle h. The function deletes the object without requesting verification even if the object is a window.

delete('filename') is the function form of delete. Use this form when the filename is stored in a string.

Note MATLAB does not ask for confirmation when you enter the delete command. To avoid accidentally losing files or graphics objects that you need, make sure that you have accurately specified the items you want deleted.

Remarks
The action that the delete function takes on deleted files depends upon the setting of the MATLAB recycle state. If you set the recycle state to on, MATLAB moves deleted files to your recycle bin or temporary directory. With the recycle state set to off (the default), deleted files are permanently removed from the system.

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.

The delete function deletes files and handles to graphics objects only. Use the rmdir function to delete directories.
Examples

To delete all files with a .mat extension in the ../mytests/ directory, type

```matlab
delete('../mytests/*\.mat')
```

To delete a directory, use rmdir rather than delete:

```matlab
rmdir mydirectory
```

See Also

recycle, dir, edit, fileparts, mkdir, rmdir, type
**Purpose**  
Delete file on FTP server

**Syntax**  
delete(f,'filename')

**Description**  
delete(f,'filename') removes the file filename from the current directory of the FTP server f, where f was created using ftp.

**Examples**  
Connect to server testsite.

```matlab
   test=ftp('ftp.testsite.com')
```

Change the current directory to testdir and view the contents.

```matlab
   cd(test,'testdir');
   dir(test)
```

**See Also**  
ftp
**Purpose**  
Remove a timer object from memory

**Syntax**  
delete(obj)

**Description**  
`delete(obj)` removes the timer object, `obj`, from memory. If `obj` is an array of timer objects, `delete` removes all the objects from memory.

When you delete a timer object, it becomes invalid and cannot be reused. Use the `clear` command to remove invalid timer objects from the workspace.

If multiple references to a timer object exist in the workspace, deleting the timer object invalidates the remaining references. Use the `clear` command to remove the remaining references to the object from the workspace.

**See Also**  
clear, isvalid, timer
demo

**Purpose**
Access product demos via Help browser

**Graphical Interface**
As an alternative to the demo function, you can select Help -> Demos from the MATLAB desktop, or click the Demos tab when the Help browser is open.

**Syntax**
demo
demo subtopic
demo subtopic category
demo('subtopic', 'category')

**Description**
demo opens the Demos panel in the Help browser. In the left pane, expand the listing for a product area (for example, MATLAB). Within that product area, expand the listing for a product or product category (for example, MATLAB Graphics). Select a specific demo from the list (for example, Visualizing Sound). In the right pane, view instructions for using the demo. For more information, see Demos in the Help Browser. To run a demo from the command line, type the demo name. For published M-file demos, that is those demos in which the H1 line begins with two comment symbols (%%), type playshow followed by the demo name to run it.

demo subtopic opens the Demos panel in the Help browser with the specified subtopic expanded. Subtopics are matlab, toolbox, simulink, and blockset.

demo subtopic product opens the Demos panel in the Help browser to the specified product or category within the subtopic. The demo function uses the full name displayed in the Demo panel for product.

demo('subtopic', 'category') is the function form of the syntax. Use this form when category is more than one word.
Examples

Accessing Toolbox Demos

To find the demos relating to the Communications Toolbox, type

demo toolbox communications

The Help browser opens to the Demos panel with the Toolbox subtopic expanded and with the Communications product highlighted and expanded to show the available demos.
**Accessing Simulink Demos**

To access the demos within Simulink, type

```
demo simulink automotive
```

The **Demos** panel opens with the Simulink subtopic and Automotive category expanded.

**Function Form of demo**

To access the Simulink Report Generator demos, run

```
demo('simulink', 'simulink report generator')
```

which displays

---

![Simulink Report Generator Demos](image)

**Simulink Report Generator Demos**

The Simulink Report Generator provides a set of flexible, customizable components which document Simulink and Stateflow projects.

---

2-594
Running a Demo from the Command Line
Type
   vibes
to run a visualization demonstration showing an animated L-shaped membrane.

Running a Published M-File Demo from the Command Line
Type
   quake
to run an earthquake data demo. Not much appears to happen. This is because quake is a published M-file demo. Verify this by viewing the M-file, quake.m, for example, by typing
   edit quake
The first line, that is, the H1 line for quake is
   %% Loma Prieta Earthquake
The %% indicates that quake is a published M-file demo. So to run it, type
   playshow quake
and the earthquake demo runs.

See Also
    help, helpbrowser, helpwin, lookfor, playshow
**Purpose**
List the dependent directories of an M-file or P-file

**Syntax**
```matlab
list = depdir('file_name');

[list,prob_files,prob_sym,prob_strings] = depdir('file_name');

[...] = depdir('file_name1','file_name2',...);
```

**Description**
The `depdir` function lists the directories of all the functions that a specified M-file or P-file needs to operate. This function is useful for finding all the directories that need to be included with a run-time application and for determining the run-time path.

`list = depdir('file_name')` creates a cell array of strings containing the directories of all the M-files and P-files that `file_name.m` or `file_name.p` uses. This includes the second-level files that are called directly by `file_name`, as well as the third-level files that are called by the second-level files, and so on.

`[list,prob_files,prob_sym,prob_strings] = depdir('file_name')` creates three additional cell arrays containing information about any problems with the `depdir` search. `prob_files` contains filenames that `depdir` was unable to parse. `prob_sym` contains symbols that `depdir` was unable to find. `prob_strings` contains callback strings that `depdir` was unable to parse.

`[...] = depdir('file_name1','file_name2',...)` performs the same operation for multiple files. The dependent directories of all files are listed together in the output cell arrays.

**Example**
```matlab
list = depdir('mesh')
```

**See Also**
depfun
**Purpose**

List the dependent functions of an M-file or P-file

**Syntax**

```matlab
list = depfun('file_name');

[list,builtins,classes] = depfun('file_name');

[list,builtins,classes,prob_files,prob_sym,eval_strings,...
called_from,java_classes] = depfun('file_name');

[...] = depfun('file_name1','file_name2',...);

[...] = depfun('fig_file_name');

[...] = depfun(...,'-toponly');
```

**Description**

The *depfun* function lists all the functions and scripts, as well as built-in functions, that a specified M-file needs to operate. This is useful for finding all of the M-files that you need to compile for a MATLAB run-time application.

`list = depfun('file_name')` creates a cell array of strings containing the paths of all the files that `file_name.m` uses. This includes the second-level files that are called directly by `file_name.m`, as well as the third-level files that are called by the second-level files, and so on.

**Note** If `depfun` reports that “These files could not be parsed:” or if the `prob_files` output below is nonempty, then the rest of the output of `depfun` might be incomplete. You should correct the problematic files and invoke `depfun` again.

```matlab
[list,builtins,classes] = depfun('file_name') creates three cell arrays containing information about dependent functions. list contains the paths of all the files that `file_name` and its subordinates use. builtins contains the built-in functions that `file_name` and its subordinates use. classes contains the MATLAB classes that `file_name` and its subordinates use.

[list,builtins,classes,prob_files,prob_sym,eval_strings,...
called_from,java_classes] = depfun('file_name') creates additional cell arrays or structure arrays containing information about any problems with the `depfun` search and about where the functions in list are invoked. The additional outputs are
- prob_files, which indicates which files depfun was unable to parse, find, or access. Parsing problems can arise from MATLAB syntax errors. prob_files is a structure array whose fields are
  - name, which gives the names of the files
  - listindex, which tells where the files appeared in list
  - errmsg, which describes the problems
- prob_sym, which indicates which symbols depfun was unable to resolve as functions or variables. It is a structure array whose fields are
  - fcn_id, which tells where the files appeared in list
  - name, which gives the names of the problematic symbols
- eval_strings, which indicates usage of these evaluation functions: eval, evalc, evalin, feval. When preparing a run-time application, you should examine this output to determine whether an evaluation function invokes a function that does not appear in list. The output eval_strings is a structure array whose fields are
  - fcn_name, which give the names of the files that use evaluation functions
  - lineno, which gives the line numbers in the files where the evaluation functions appear
- called_from, a cell array of the same length as list. This cell array is arranged so that
  list(called_from{i})
  returns all functions in file_name that invoke the function list{i}.
- java_classes, a cell array of Java class names that file_name and its subordinates use

[... ] = depfun('file_name1', 'file_name2', ...) performs the same operation for multiple files. The dependent functions of all files are listed together in the output arrays.

[... ] = depfun('fig_file_name') looks for dependent functions among the callback strings of the GUI elements that are defined in the .fig or .mat file named fig_file_name.

[... ] = depfun(...,'-toponly') differs from the other syntaxes of depfun in that it examines only the files listed explicitly as input arguments. It does
not examine the files on which they depend. In this syntax, the flag `-toponly' must be the last input argument.

Notes

1 If `depfun` does not find a file called `hginfo.mat` on the path, then it creates one. This file contains information about Handle Graphics callbacks.

2 If your application uses toolbar items from the MATLAB default figure window, then you must include `FigureToolBar.fig` in your input to `depfun`.

3 If your application uses menu items from the MATLAB default figure window, then you must include `FigureMenuBar.fig` in your input to `depfun`.

4 Because many built-in Handle Graphics functions invoke `newplot`, the list produced by `depfun` always includes the functions on which `newplot` is dependent:

   - `matlabroot\toolbox\matlab\graphics\newplot.m`
   - `matlabroot\toolbox\matlab\graphics\closereq.m`
   - `matlabroot\toolbox\matlab\graphics\gcf.m`
   - `matlabroot\toolbox\matlab\graphics\gca.m`
   - `matlabroot\toolbox\matlab\graphics\private\clo.m`
   - `matlabroot\toolbox\matlab\general\@char\delete.m`
   - `matlabroot\toolbox\matlab\lang\nargchk.m`
   - `matlabroot\toolbox\matlab\uitools\allchild.m`
   - `matlabroot\toolbox\matlab\ops\setdiff.m`
   - `matlabroot\toolbox\matlab\ops\@cell\setdiff.m`
   - `matlabroot\toolbox\matlab\iofun\filesep.m`
   - `matlabroot\toolbox\matlab\ops\unique.m`
   - `matlabroot\toolbox\matlab\elmat\repmat.m`
   - `matlabroot\toolbox\matlab\datafun\sortrows.m`
   - `matlabroot\toolbox\matlab\strfun\deblank.m`
   - `matlabroot\toolbox\matlab\ops\@cell\unique.m`
   - `matlabroot\toolbox\matlab\strfun\@cell\deblank.m`
   - `matlabroot\toolbox\matlab\datafun\@cell\sort.m`
   - `matlabroot\toolbox\matlab\strfun\cellstr.m`
   - `matlabroot\toolbox\matlab\datatypes\iscell.m`
   - `matlabroot\toolbox\matlab\strfun\iscellstr.m`
depfun

- 'matlabroot\toolbox\matlab\datatypes\cellfun.dll'

Examples

list = depfun('mesh'); % Files mesh.m depends on
list = depfun('mesh','-toponly') % Files mesh.m depends on
directly
[list,builtins,classes] = depfun('gca');

See Also
depdir, profile
Purpose
Matrix determinant

Syntax
\[ d = \text{det}(X) \]

Description
\( d = \text{det}(X) \) returns the determinant of the square matrix \( X \). If \( X \) contains only integer entries, the result \( d \) is also an integer.

Remarks
Using \( \text{det}(X) == 0 \) as a test for matrix singularity is appropriate only for matrices of modest order with small integer entries. Testing singularity using \( \text{abs} \left( \text{det}(X) \right) \leq \text{tolerance} \) is not recommended as it is difficult to choose the correct tolerance. The function \( \text{cond}(X) \) can check for singular and nearly singular matrices.

Algorithm
The determinant is computed from the triangular factors obtained by Gaussian elimination

\[
\begin{align*}
[L, U] &= \text{lu}(A) \\
S &= \text{det}(L) & \text{This is always +1 or -1} \\
\text{det}(A) &= S \times \text{prod(diag(U))}
\end{align*}
\]

Examples
The statement \( A = \begin{bmatrix} 1 & 2 & 3; 4 & 5 & 6; 7 & 8 & 9 \end{bmatrix} \)
produces
\[
A = \\
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\]

This happens to be a singular matrix, so \( d = \text{det}(A) \) produces \( d = 0 \).
Changing \( A(3,3) \) with \( A(3,3) = 0 \) turns \( A \) into a nonsingular matrix. Now \( d = \text{det}(A) \) produces \( d = 27 \).

See Also
\( \text{cond}, \text{condest}, \text{inv}, \text{lu}, \text{rref} \)

The arithmetic operators \( \backslash, / \)


**Purpose**  
Remove linear trends.

**Syntax**  
\[ y = \text{detrend}(x) \]
\[ y = \text{detrend}(x, 'constant') \]
\[ y = \text{detrend}(x, 'linear', bp) \]

**Description**  
\text{detrend} removes the mean value or linear trend from a vector or matrix, usually for FFT processing.

\[ y = \text{detrend}(x) \] removes the best straight-line fit from vector \( x \) and returns it in \( y \). If \( x \) is a matrix, \text{detrend} removes the trend from each column.

\[ y = \text{detrend}(x, 'constant') \] removes the mean value from vector \( x \) or, if \( x \) is a matrix, from each column of the matrix.

\[ y = \text{detrend}(x, 'linear', bp) \] removes a continuous, piecewise linear trend from vector \( x \) or, if \( x \) is a matrix, from each column of the matrix. Vector \( bp \) contains the indices of the breakpoints between adjacent linear segments. The breakpoint between two segments is defined as the data point that the two segments share.

\[ \text{detrend}(x, 'linear'), \text{with no breakpoint vector specified, is the same as detrend}(x). \]

**Example**
\[
\begin{align*}
\text{sig} & = [0 \ 1 \ -2 \ 1 \ 0 \ 1 \ -2 \ 1 \ 0]; \quad \% \text{signal with no linear trend} \\
\text{trend} & = [0 \ 1 \ 2 \ 3 \ 4 \ 3 \ 2 \ 1 \ 0]; \quad \% \text{two-segment linear trend} \\
\text{x} & = \text{sig} + \text{trend}; \quad \% \text{signal with added trend} \\
\text{y} & = \text{detrend}(\text{x}, 'linear', 5) \quad \% \text{breakpoint at 5th element}
\end{align*}
\]
y =

-0.0000
1.0000
-2.0000
1.0000
0.0000
1.0000
-2.0000
1.0000
-0.0000

Note that the breakpoint is specified to be the fifth element, which is the data point shared by the two segments.

Algorithm  detrend computes the least-squares fit of a straight line (or composite line for piecewise linear trends) to the data and subtracts the resulting function from the data. To obtain the equation of the straight-line fit, use polyfit.

See Also  polyfit
**Purpose**
Evaluate the solution of a differential equation

**Syntax**
\[
\begin{align*}
\text{sxint} &= \text{deval}(\text{sol},\text{xint}) \\
\text{sxint} &= \text{deval}(\text{xint},\text{sol}) \\
\text{sxint} &= \text{deval}(\text{sol},\text{xint},\text{idx}) \\
\text{sxint} &= \text{deval}(\text{xint},\text{sol},\text{idx}) \\
[\text{sxint}, \text{spxint}] &= \text{deval}(\ldots)
\end{align*}
\]

**Description**
\[
\text{sxint} = \text{deval}(\text{sol},\text{xint}) \text{ and } \text{sxint} = \text{deval}(\text{xint},\text{sol}) \text{ evaluate the solution of a differential equation problem. sol is a structure returned by one of these solvers:}
\]
- An initial value problem solver (ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb, ode15i)
- The delay differential equations solver (dde23),
- The boundary value problem solver (bvp4c).

\text{xint} is a point or a vector of points at which you want the solution. The elements of \text{xint} must be in the interval [\text{sol.x(1)}, \text{sol.x(end)}]. For each \text{i}, \text{sxint(:,i)} is the solution at \text{xint(i)}.

\[
\begin{align*}
\text{sxint} &= \text{deval}(\text{sol},\text{xint},\text{idx}) \text{ and } \text{sxint} = \text{deval}(\text{xint},\text{sol},\text{idx}) \text{ evaluate as above but return only the solution components with indices listed in the vector idx.}
\end{align*}
\]

\[
[\text{sxint}, \text{spxint}] = \text{deval}(\ldots) \text{ also returns spxint, the value of the first derivative of the polynomial interpolating the solution.}
\]

**Note**
For multipoint boundary value problems, the solution obtained by bvp4c might be discontinuous at the interfaces. For an interface point xc, deval returns the average of the limits from the left and right of xc. To get the limit values, set the \text{xint} argument of deval to be slightly smaller or slightly larger than xc.

**Example**
This example solves the system \(y' = vdp1(t,y)\) using ode45, and evaluates and plots the first component of the solution at 100 points in the interval \([0,20]\).
sol = ode45(@vdp1,[0 20],[2 0]);
x = linspace(0,20,100);
y = deval(sol,x,1);
plot(x,y);

See Also

- ODE solvers: ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb, ode15i
- DDE solver: dde23
- BVP solver: bvp4c
Purpose

Diagonal matrices and diagonals of a matrix

Syntax

\[ X = \text{diag}(v, k) \]
\[ X = \text{diag}(v) \]
\[ v = \text{diag}(X, k) \]
\[ v = \text{diag}(X) \]

Description

\[ X = \text{diag}(v, k) \] when \( v \) is a vector of \( n \) components, returns a square matrix \( X \) of order \( n + \text{abs}(k) \), with the elements of \( v \) on the \( k \)th diagonal. \( k = 0 \) represents the main diagonal, \( k > 0 \) above the main diagonal, and \( k < 0 \) below the main diagonal.

\[ X = \text{diag}(v) \] puts \( v \) on the main diagonal, same as above with \( k = 0 \).

\[ v = \text{diag}(X, k) \] for matrix \( X \), returns a column vector \( v \) formed from the elements of the \( k \)th diagonal of \( X \).

\[ v = \text{diag}(X) \] returns the main diagonal of \( X \), same as above with \( k = 0 \).

Examples

\[ \text{diag} \left( \text{diag}(X) \right) \] is a diagonal matrix.

\[ \text{sum} \left( \text{diag}(X) \right) \] is the trace of \( X \).

The statement

\[ \text{diag}(-m:m)+\text{diag(ones(2*m,1),1)}+\text{diag(ones(2*m,1),-1)} \]

produces a tridiagonal matrix of order \( 2*m+1 \).
See Also

spdiags, tril, triu
Purpose

Create and display dialog box

Syntax

\[ h = \text{dialog('PropertyName', PropertyValue, ...)} \]

Description

\[ h = \text{dialog('PropertyName', PropertyValue, ...)} \] returns a handle to a dialog box. This function creates a figure graphics object and sets the figure properties recommended for dialog boxes. You can specify any valid figure property value.

See Also

erordlg, figure, helpdlg, inputdlg, pagesetupdlg, printdlg, questdlg, uiwait, uiresume, warndlg

“Predefined Dialog Boxes” for related functions
**Purpose**

Save session to a file

**Syntax**

```plaintext
diary

\[\text{filename}\]

diary~off

\text{on}

\text{filename}
```

**Description**

The `diary` function creates a log of keyboard input and the resulting text output, with some exceptions (see “Remarks” for details). The output of `diary` is an ASCII file, suitable for searching in, printing, inclusion in most reports and other documents. If you do not specify `filename`, MATLAB creates a file named `diary` in the current directory.

`diary` toggles `diary` mode on and off. To see the status of `diary`, type `get(0, 'Diary')`. MATLAB returns either `on` or `off` indicating the `diary` status.

`diary('filename')` writes a copy of all subsequent keyboard input and the resulting output (except it does not include graphics) to the named file, where `filename` is the full pathname or `filename` is in the current MATLAB directory. If the file already exists, output is appended to the end of the file. You cannot use a `filename` called `off` or `on`. To see the name of the `diary` file, use `get(0, 'DiaryFile')`.

`diary~off` suspends the `diary`.

`diary~on` resumes `diary` mode using the current `filename`, or the default `filename` `diary` if none has yet been specified.

`diary~filename` is the unquoted form of the syntax.

**Remarks**

Because the output of `diary` is plain text, the file does not exactly mirror input and output from the Command Window:

- Output does not include graphics (figure windows).
- Syntax highlighting and font preferences are not preserved.
Hidden components of Command Window output such as hyperlink information generated with `matlab:` are shown in plain text. For example, if you enter the following statement:

```matlab
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

MATLAB displays

Generate magic square

However, the diary file, when viewed in a text editor, shows:

```matlab
disp('<a href="matlab:magic(4)">Generate magic square</a>')

Generate magic square
```

If you view the output of `diary` in the Command Window, the Command Window interprets the `<a href ...>` statement and displays it as a hyperlink.

Viewing the output of `diary` in a console window might produce different results compared to viewing `diary` output in the desktop Command Window. One example is using the `\r` option for the `fprintf` function; using the `\n` option might alleviate that problem.

**See Also**

Command History in MATLAB Desktop Tools documentation
Purpose

Differences and approximate derivatives

Syntax

\[ Y = \text{diff}(X) \]
\[ Y = \text{diff}(X,n) \]
\[ Y = \text{diff}(X,n,\text{dim}) \]

Description

\( Y = \text{diff}(X) \) calculates differences between adjacent elements of \( X \).

If \( X \) is a vector, then \( \text{diff}(X) \) returns a vector, one element shorter than \( X \), of differences between adjacent elements:

\[
[X(2) - X(1) \ X(3) - X(2) \ldots \ X(n) - X(n - 1)]
\]

If \( X \) is a matrix, then \( \text{diff}(X) \) returns a matrix of row differences:

\[
[X(2:m,:) - X(1:m-1,:),]
\]

In general, \( \text{diff}(X) \) returns the differences calculated along the first non-singleton \( \text{size}(X,\text{dim}) > 1 \) dimension of \( X \).

\( Y = \text{diff}(X,n) \) applies \( \text{diff} \) recursively \( n \) times, resulting in the \( n \)th difference. Thus, \( \text{diff}(X,2) \) is the same as \( \text{diff}(\text{diff}(X)) \).

\( Y = \text{diff}(X,n,\text{dim}) \) is the \( n \)th difference function calculated along the dimension specified by scalar \( \text{dim} \). If order \( n \) equals or exceeds the length of dimension \( \text{dim} \), \( \text{diff} \) returns an empty array.

Remarks

Since each iteration of \( \text{diff} \) reduces the length of \( X \) along dimension \( \text{dim} \), it is possible to specify an order \( n \) sufficiently high to reduce \( \text{dim} \) to a singleton \( \text{size}(X,\text{dim}) = 1 \) dimension. When this happens, \( \text{diff} \) continues calculating along the next nonsingleton dimension.

Examples

The quantity \( \text{diff}(y) ./ \text{diff}(x) \) is an approximate derivative.

\[
x = [1 \ 2 \ 3 \ 4 \ 5];
y = \text{diff}(x)
y =
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1
\end{bmatrix}
\]

\[
z = \text{diff}(x,2)
z =
\]

2-611
Given,

\[ A = \text{rand}(1,3,2,4); \]

diff(A) is the first-order difference along dimension 2.
diff(A,3,4) is the third-order difference along dimension 4.

**See Also**
gradient, prod, sum
Purpose
Display directory listing

Graphical Interface
As an alternative to the `dir` function, use the Current Directory browser.

Syntax
```matlab
dir
dir name
files = dir('name')
```

Description
`dir` lists the files in the current working directory. Results are not sorted, but presented in the order returned by the operating system.

`dir name` lists the specified files. The `name` argument can be a pathname, filename, or can include both. You can use absolute and relative pathnames and wildcards (*).

`files = dir('directory')` returns the list of files in the specified directory (or the current directory, if `dirname` is not specified) to an m-by-1 structure with the fields

- `name`: Filename
- `date`: Modification date
- `bytes`: Number of bytes allocated to the file
- `isdir`: 1 if name is a directory; 0 if not

Examples

List Directory Contents
To view the contents of the `matlab/audio` directory, type
```
dir $matlabroot/toolbox/matlab/audio
```

Using Wildcard and File Extension
To view the MAT files in your current working directory that include the term `java`, type
```
dir *java*.mat
```
MATLAB returns
```
java_array.mat javafrmobj.mat testjava.mat
```
Using Relative Pathname
To view the M-files in the MATLAB audio directory, type

```matlab
dir(fullfile(matlabroot,'toolbox/matlab/audio/*.m'))
```

MATLAB returns:

```
Contents.m  auread.m  soundsc.m
audiodevinfo.m  auwrite.m  wavplay.m
audioplayer.m  lin2mu.m  wavread.m
audioplayerreg.m  mu2lin.m  wavrecord.m
audiorecorder.m  prefspanel.m  wavwrite.m
audiouniquename.m  sound.m
```

Returning File List to Structure
To return the list of files to the variable audio_files, type

```matlab
audio_files=dir(fullfile(matlabroot,'toolbox/matlab/audio/*.m'))
```

MATLAB returns the information in a structure array.

```matlab
audio_files = 
19x1 struct array with fields:
    name
    date
    bytes
    isdir
```

Index into the structure to access a particular item. For example,

```matlab
audio_files(3).name
ans =
    audioplayer.m
```

See Also

`cd`, `copyfile`, `delete`, `fileattrib`, `filebrowser`, `fileparts`, `isdir`, `ls`, `matlabroot`, `mkdir`, `mfilename`, `movefile`, `rmdir`, `type`, `what`
Purpose
List contents of directory on FTP server

Syntax
dir(f,'dirname')
d=dir(…)

Description
dir(f,'dirname') lists the files in the specified directory, dirname, on the FTP server f, where f was created using ftp. If dirname is unspecified, dir lists the files in the current directory of f.

d=dir(…) returns the results in an m-by-1 structure with the following fields for each file:

<table>
<thead>
<tr>
<th>name</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Date last modified</td>
</tr>
<tr>
<td>bytes</td>
<td>Size of the file</td>
</tr>
<tr>
<td>isdir</td>
<td>1 if name is a directory and 0 if not</td>
</tr>
</tbody>
</table>

Examples
Connect to the MathWorks FTP server and view the contents.
```
tmw=ftp('ftp.mathworks.com');
dir(tmw)
```
```
.         incoming    pickup
README    matlab        pub
README.incoming outgoing  pubs
```
Change to the directory pub/pentium.
```
 cd(tmw,'pub/pentium')
```
dir (ftp)

View the contents of that directory.

dir(tmw)

.                    Intel_resp.txt        NYT_2.txt
..                    Intel_support.txt     NYT_Dec14.uu
Andy_Grove.txt       Intel_white.ps       New_York_Times.txt
Associated_Press.txt MathWorks_press.txt Nicely_1.txt
CNN.html             Mathisen.txt          Nicely_2.txt
Coe.txt              Moler_1.txt           Nicely_3.txt
Cygnus.txt           Moler_2.txt           Pratt.txt
EE_Times.txt         Moler_3.txt           README.txt
FAQ.txt              Moler_4.txt           SPSS.txt
IBM_study.txt        Moler_5.txt           Smith.txt
Intel_FAX.txt         Moler_6.ps            p87test.txt
Intel_fix.txt         Moler_7.txt           p87test.zip
Intel_replace.txt     Myths.txt             test

Or return the results to the structure m.

m=dir(tmw)

m =

37x1 struct array with fields:
    name
    date
    bytes
    isdir

View element 17.

m(17)

ans =

    name: 'Moler_1.txt'
    date: '1995 Mar  27'
    bytes: 3427
    isdir: 0

See Also    ftp, mkdir (ftp), rmdir (ftp)
Purpose
Display text or array

Syntax
disp(X)

Description
disp(X) displays an array, without printing the array name. If X contains a text string, the string is displayed.

Another way to display an array on the screen is to type its name, but this prints a leading X =, which is not always desirable.

Note that disp does not display empty arrays.

Examples
One use of disp in an M-file is to display a matrix with column labels:

```
disp('     Corn     Oats     Hay')
disp(rand(5,3))
```

which results in

```
     Corn     Oats     Hay
 0.2113     0.8474     0.2749
 0.0820     0.4524     0.8807
 0.7599     0.8075     0.6538
 0.0087     0.4832     0.4899
 0.8096     0.6135     0.7741
```

See Also
format, int2str, num2str, rats, sprintf
**disp (timer)**

**Purpose**
Display information about timer object

**Syntax**

```
obj
disp(obj)
```

**Description**

`obj` or `disp(obj)` displays summary information for the timer object, `obj`. If `obj` is an array of timer objects, `disp` outputs a table of summary information about the timer objects in the array.

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when

- Creating a timer object, using the `timer` function
- Configuring property values using the dot notation

**Examples**
The following commands display summary information for timer object `t`.

```
t = timer

Timer Object: timer-1

Timer Settings
  ExecutionMode: singleShot
  Period: 1
  BusyMode: drop
  Running: off

Callbacks
  TimerFcn: []
  ErrorFcn: []
  StartFcn: []
  StopFcn: []
```

This example shows the format of summary information displayed for an array of timer objects.

```
t2 = timer;
disp(timerfind)
```

Timer Object Array
disp (timer)

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>
</tbody>
</table>

See Also

timer, get
**display**

**Purpose**
Overloaded method to display an object

**Syntax**
display(X)

**Description**
display(X) prints the value of a variable or expression, X. MATLAB calls display(X) when it interprets a variable or expression, X, that is not terminated by a semicolon. For example, sin(A) calls display, while sin(A); does not.

If X is an instance of a MATLAB class, then MATLAB calls the display method of that class, if such a method exists. If the class has no display method or if X is not an instance of a MATLAB class, then the MATLAB built-in display function is called.

**Examples**
A typical implementation of display calls disp to do most of the work and looks like this.

```matlab
function display(X)
    if isequal(get(0,'FormatSpacing'),'compact')
        disp([inputname(1) ' =']);
        disp(X)
    else
        disp(' ')
        disp([inputname(1) ' =']);
        disp(' ');
        disp(X)
    end
```

The expression magic(3), with no terminating semicolon, calls this function as display(magic(3)).

```matlab
magic(3)
```

ans =

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

As an example of a class display method, the function below implements the display method for objects of the MATLAB class polynom.
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom
    disp(' ');
    disp(['inputname(1),'' = ''])
    disp(' ');
    disp([' '' char(p)])
    disp(' ');

The statement
    p = polynom([1 0 -2 -5])

creates a polynom object. Since the statement is not terminated with a
semicolon, the MATLAB interpreter calls display(p), resulting in the output

    p =
    x^3 - 2*x - 5

See Also    disp, ans, sprintf, special characters
### divergence

**Purpose**

Computes the divergence of a vector field

**Syntax**

- `div = divergence(X,Y,Z,U,V,W)`
- `div = divergence(U,V,W)`
- `div = divergence(X,Y,U,V)`
- `div = divergence(U,V)`

**Description**

- `div = divergence(X,Y,Z,U,V,W)` computes the divergence of a 3-D vector field `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`).

- `div = divergence(U,V,W)` assumes `X, Y, Z` are determined by the expression

  ![Math expression]

  where `[m,n,p] = size(U)`.

- `div = divergence(X,Y,U,V)` computes the divergence of a 2-D vector field `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`).

- `div = divergence(U,V)` assumes `X` and `Y` are determined by the expression

  ![Math expression]

  where `[m,n] = size(U)`.

**Examples**

This example displays the divergence of vector volume data as slice planes using color to indicate divergence.

```matlab
load wind
div = divergence(x,y,z,u,v,w);
slice(x,y,z,div,[90 134],[59],[0]);
shading interp
daspect([1 1 1])
camlight
```
See Also

streamtube, curl, isosurface

“Volume Visualization” for related functions

Displaying Divergence with Stream Tubes for another example
**dlmread**

**Purpose**
Read an ASCII-delimited file into a matrix

**Graphical Interface**
As an alternative to `dlmread`, use the Import Wizard. To activate the Import Wizard, select **Import data** from the **File** menu.

**Syntax**

\[
M = dlmread('filename')
\]

\[
M = dlmread('filename', delimiter)
\]

\[
M = dlmread('filename', delimiter, R, C)
\]

\[
M = dlmread('filename', delimiter, range)
\]

**Description**

\(M = dlmread('filename')\) reads numeric data from the ASCII-delimited file `filename`, using a delimiter inferred from the formatting of the file. Comma (,) is the default delimiter.

\(M = dlmread('filename', delimiter)\) reads numeric data from the ASCII-delimited file `filename`, using the specified delimiter. Use \(\backslash t\) to specify a tab delimiter.

\(M = dlmread('filename', delimiter, R, C)\) reads numeric data from the ASCII-delimited file `filename`, using the specified delimiter. The values \(R\) and \(C\) specify the row and column where the upper left corner of the data lies in the file. \(R\) and \(C\) are zero based, so that \(R=0, C=0\) specifies the first value in the file, which is the upper left corner.

\(M = dlmread('filename', delimiter, range)\) reads the range specified by `range = [R1 C1 R2 C2]` where \((R1,C1)\) is the upper left corner of the data to be read and \((R2,C2)\) is the lower right corner. You can also specify the range using spreadsheet notation, as in `range = 'A1..B7'`. 

---

**Note**
When a delimiter is inferred from the formatting of the file, consecutive whitespaces are treated as a single delimiter. By contrast, if a delimiter is specified by the `delimiter` input, any repeated delimiter character is treated as a separate delimiter.

\(M = dlmread('filename', delimiter, R, C)\) reads numeric data from the ASCII-delimited file `filename`, using the specified delimiter. The values \(R\) and \(C\) specify the row and column where the upper left corner of the data lies in the file. \(R\) and \(C\) are zero based, so that \(R=0, C=0\) specifies the first value in the file, which is the upper left corner.
dlmread fills empty delimited fields with zero. Data files having lines that end with a non-space delimiter, such as a semicolon, produce a result that has an additional last column of zeros.

dlmread 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;-&lt;real&gt;-&lt;imag&gt;i</td>
<td>j</td>
</tr>
<tr>
<td>&lt;-&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.

**See Also**
dlmwrite, textscan, csvread, csvwrite, wk1read, wk1write
dlmwrite

**Purpose**

Write a matrix to an ASCII-delimited file

**Syntax**

```
dlmwrite('filename', M)
dlmwrite('filename', M, 'D')
dlmwrite('filename', M, 'D', R, C)
dlmwrite('filename', M, attribute1, value1, attribute2, value2, ...)
dlmwrite('filename', M, '-append')
dlmwrite('filename', M, '-append', attribute-value list)
```

**Description**

`dlmwrite('filename', M)` writes matrix `M` into an ASCII format file using the default delimiter (,) to separate matrix elements. The data is written starting at the first column of the first row in the destination file, `filename`.

`dlmwrite('filename', M, 'D')` writes matrix `M` into an ASCII format file, using delimiter `D` to separate matrix elements. The data is written starting at the first column of the first row in the destination file, `filename`. A comma (,) is the default delimiter. Use \t to produce tab-delimited files.

`dlmwrite('filename', M, 'D', R, C)` writes matrix `M` into an ASCII format file, using delimiter `D` to separate matrix elements. The data is written starting at row `R` and column `C` in the destination file, `filename`. `R` and `C` are zero based, so that `R=0, C=0` specifies the first value in the file, which is the upper left corner.

`dlmwrite('filename', M, 'attrib1', value1, 'attrib2', value2, ...)` is an alternate syntax to those shown above, in which you specify any number of attribute-value pairs in any order in the argument list. Each attribute must be immediately followed by a corresponding value (see the table below).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>delimiter</td>
<td>Delimiter string to be used in separating matrix elements</td>
</tr>
<tr>
<td>newline</td>
<td>Character(s) to use in terminating each line (see table below)</td>
</tr>
<tr>
<td>roffset</td>
<td>Offset, in rows, from the top of the destination file to where matrix data is to be written. Offset is zero based.</td>
</tr>
</tbody>
</table>
This table shows which values you can use when setting the newline attribute.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>offset</strong></td>
<td>Offset, in columns, from the left side of the destination file to where matrix data is to be written. Offset is zero based.</td>
</tr>
<tr>
<td><strong>precision</strong></td>
<td>Numeric precision to use in writing data to the file. Specify the number of significant digits or a C-style format string starting in %, such as '%10.5f'.</td>
</tr>
</tbody>
</table>

Remarks
The resulting file is readable by spreadsheet programs.

Examples
Export matrix M to a file delimited by the tab character and using a precision of six significant digits:

```matlab
dlmwrite('myfile.txt', M, 'delimiter', '\t', 'precision', 6)
type myfile.txt
```

```
0.893898   0.284409   0.582792   0.432907
0.199138   0.469224   0.423496   0.22595
0.298723   0.0647811  0.515512   0.579807
0.661443   0.988335   0.333951   0.760365
```
Export matrix \( M \) to a file using a precision of six decimal places and the conventional line terminator for the PC platform:

```matlab
dlmwrite('myfile.txt', m, 'precision', '%.6f', 'newline', 'pc')
type myfile.txt
```

```
16.000000, 2.000000, 3.000000, 13.000000
  5.000000, 11.000000, 10.000000, 8.000000
  9.000000, 7.000000, 6.000000, 12.000000
  4.000000, 14.000000, 15.000000, 1.000000
```

Export matrix \( M \) to a file, and then append an additional matrix to the file that is offset one row below the first:

```matlab
M = magic(4);
dlmwrite('myfile.txt', [M*5 M/5], ' ')
dlmwrite('myfile.txt', rand(3), 'append', 'on', ...
  'roffset', 1, 'delimiter', ' ')
type myfile.txt
```

```
80 10 15 65 3.2 0.4 0.6 2.6
25 55 50 40 1 2.2 2 1.6
45 35 30 60 1.8 1.4 1.2 2.4
20 70 75 5 0.8 2.8 3 0.2

0.99008 0.49831 0.32004
0.78886 0.21396 0.9601
0.43866 0.64349 0.72663
```

**See Also**
dlmread, csvwrite, csvread, wk1write, wk1read
Dulmage-Mendelsohn decomposition

$p = \text{dmperm}(A)$ if $A$ is square and has full rank, returns a row permutation $p$ so that $A(p,:)$ has nonzero diagonal elements. This permutation is also called a perfect matching. If $A$ is not square or not full rank, $p$ is a vector that identifies a matching of maximum size: for each column $j$ of $A$, either $p(j)=0$ or $A(p(j),j)$ is nonzero.

$[p,q,r,s] = \text{dmperm}(A)$, where $A$ need not be square or full rank, finds permutations $p$ and $q$ and index vectors $r$ and $s$ so that $A(p,q)$ is block upper triangular. The $k$th block has indices $(r(k):r(k+1)-1, s(k):s(k+1)-1)$. When $A$ is square and has full rank, $r = s$.

If $A$ is not square or not full rank, the first block may have more columns and the last block may have more rows. All other blocks are square and irreducible. $\text{dmperm}$ permutes nonzeros to the diagonals of square blocks, but does not do this for non-square blocks.

If $A$ is a reducible matrix, the linear system $Ax = b$ can be solved by permuting $A$ to a block upper triangular form, with irreducible diagonal blocks, and then performing block backsubstitution. Only the diagonal blocks of the permuted matrix need to be factored, saving fill and arithmetic in the blocks above the diagonal.

In graph theoretic terms, $\text{dmperm}$ finds a maximum-size matching in the bipartite graph of $A$, and the diagonal blocks of $A(p,q)$ correspond to the strong Hall components of that graph. The output of $\text{dmperm}$ can also be used to find the connected or strongly connected components of an undirected or directed graph. For more information see Pothen and Fan [1].

$sparkin$

Purpose
Display online documentation in MATLAB Help browser

Graphical Interface
As an alternative to the doc function, use the Help browser Search tab. Type the function name and click Go.

Syntax

doc

doc functionname

doc toolboxname/

doc toolboxname/functionname

Description
doc opens the Help browser, if it is not already running, or brings the window on top when it is already open.

doc functionname displays the reference page for the MATLAB function functionname in the Help browser (for example, you are looking at the reference page for the doc function). If functionname is overloaded, that is, if functionname appears in multiple directories on the MATLAB search path, doc displays the reference page for the first functionname on the search path and displays a hyperlinked list of the other functions and their directories in the MATLAB Command Window. If a reference page for functionname does not exist, doc displays its M-file help in the Help browser.

doc toolboxname displays the Roadmap page for toolboxname in the Help browser, which provides a summary of the most pertinent documentation for that product.

doc toolboxname/functionname displays the reference page for functionname that belongs to the specified toolboxname, in the Help browser. This is useful for overloaded functions.

Examples
Type doc abs to display the reference page for the abs function. If Simulink and the Signal Processing Toolbox are installed and on the search path, the Command Window lists hyperlinks for the abs function in those products

```
doc signal/abs

doc simulink/abs
```

Type doc signal/abs to display the reference page for the abs function in the Signal Processing Toolbox.
Type `doc signal` to display the Roadmap page for the Signal Processing Toolbox.

**Note** If there is a function called `name` as well as a toolbox called `name`, the Roadmap page for the toolbox called `name` displays. To see the reference page for the function called `name`, use `doc toolboxname/name`, where `toolboxname` is the name of the toolbox in which the function `name` resides. For example, `doc matlab` displays the roadmap page for `matlab`, while `doc matlab/matlab` displays the reference page for the `matlab` UNIX startup function.

**See Also**
`docopt`, `docsearch`, `help`, `helpbrowser`, `lookfor`, `type`, `web`
Purpose

Web browser for UNIX platforms

Syntax
docopt

Description
docopt displays the Web browser used with MATLAB on non-Macintosh UNIX platforms, with the default being netscape (for Netscape). For non-Macintosh UNIX platforms, you can modify the docopt.m file to specify the Web browser MATLAB uses. The Web browser is used with the web function and its -browser option. It is also used for links to external Web sites from the Help.

doccmd = docopt returns a string containing the command that web -browser uses to invoke a Web browser.

To change the browser, edit the docopt.m file and change line 51. For example,

50 elseif isunix
51     doccmd = '';

Remove the comment symbol. In the quote, enter the command that launches your Web browser, and save the file. For example

51      doccmd = 'mozilla';

specifies Mozilla as the Web browser MATLAB uses.

See Also
doc, edit, helpbrowser, web
Purpose

Open Help browser Search pane and run search for specified term.

Graphical Interface

As an alternative to the docsearch function, select Desktop -> Help and click the Search tab.

Syntax

docsearch

docsearch word

docsearch ('word1 word2 ...')

docsearch('word1 word2 BOOLEANOP word3')

Description

docsearch opens the Help browser to the Search pane, or if the Help browser is already open, brings it to the top.

docsearch word1 executes a Help browser full-text search for word1, displaying results in the Help browser Search pane.

docsearch ('word1 word2 ...') executes a Help browser full-text search for pages containing word1 and word2 and any other specified words, displaying results in the Help browser Search pane.

docsearch('word1 word2 BOOLEANOP word3') executes a Help browser full-text search for the term word1 word2 BOOLEANOP word3, where BOOLEANOP is a Boolean operator (AND, NOT, OR) used to limit the search. Results display in the Help browser Search pane.

Examples

docsearch print finds all pages that contain the word print.

docsearch('print figure') finds all pages that contain the words print and figure.

docsearch('print OR printing AND figure NOT exporting') finds all pages that contain the words print and figure, or printing and figure, but only if the pages do not contain the word exporting.

See Also

doc, helpbrowser

Search Documentation with the Help Browser
dos

Purpose
Execute a DOS command and return result

Syntax
do command
status = dos('command')
[status,result] = dos('command')
[status,result] = dos('command','-echo')

Description
do command calls upon the shell to execute the given command for Windows systems.

status = dos('command') returns completion status to the status variable.

[status,result] = dos('command') in addition to completion status, returns the result of the command to the result variable.

[status,result] = dos('command','-echo') forces the output to the Command Window, even though it is also being assigned into a variable.

Both console (DOS) programs and Windows programs may be executed, but the syntax causes different results based on the type of programs. Console programs have stdout and their output is returned to the result variable. They are always run in an iconified DOS or Command Prompt Window except as noted below. Console programs never execute in the background. Also, MATLAB will always wait for the stdout pipe to close before continuing execution. Windows programs may be executed in the background as they have no stdout.

The ampersand, &, character has special meaning. For console programs this causes the console to open. Omitting this character will cause console programs to run iconically. For Windows programs, appending this character will cause the application to run in the background. MATLAB will continue processing.

Examples
The following example performs a directory listing, returning a zero (success) in s and the string containing the listing in w.

   [s, w] = dos('dir');

To open the DOS 5.0 editor in a DOS window

   dos('edit &')
To open the notepad editor and return control immediately to MATLAB

```matlab
dos('notepad file.m &')
```

The next example returns a one in `s` and an error message in `w` because `foo` is not a valid shell command.

```matlab
[s, w] = dos('foo')
```

This example echoes the results of the `dir` command to the Command Window as it executes as well as assigning the results to `w`.

```matlab
[s, w] = dos('dir', '-echo');
```

**See Also**

! (exclamation point), `perl`, `system`, `unix`, `winopen`
**Purpose**
Vector dot product

**Syntax**

```
C = dot(A,B)
C = dot(A,B,dim)
```

**Description**

`C = dot(A,B)` returns the scalar product of the vectors `A` and `B`. `A` and `B` must be vectors of the same length. When `A` and `B` are both column vectors, `dot(A,B)` is the same as `A'*B`.

For multidimensional arrays `A` and `B`, `dot` returns the scalar product along the first non-singleton dimension of `A` and `B`. `A` and `B` must have the same size.

`C = dot(A,B,dim)` returns the scalar product of `A` and `B` in the dimension `dim`.

**Examples**
The dot product of two vectors is calculated as shown:

```
a = [1 2 3]; b = [4 5 6];
c = dot(a,b)
```

```
c =
32
```

**See Also**
cross
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Convert to double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>double(X)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td><code>double(x)</code> returns the double-precision value for <code>X</code>. If <code>X</code> is already a double-precision array, <code>double</code> has no effect.</td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td><code>double</code> is called for the expressions in <code>for</code>, <code>if</code>, and <code>while</code> loops if the expression isn't already double-precision. <code>double</code> should be overloaded for any object when it makes sense to convert it to a double-precision value.</td>
</tr>
</tbody>
</table>
Purpose
Drag rectangles with mouse

Syntax
[finalrect] = dragrect(initialrect)
[finalrect] = dragrect(initialrect,stepsize)

Description
[finalrect] = dragrect(initialrect) tracks one or more rectangles anywhere on the screen. The n-by-4 matrix initialrect defines the rectangles. Each row of initialrect must contain the initial rectangle position as [left bottom width height] values. dragrect returns the final position of the rectangles in finalrect.

[finalrect] = dragrect(initialrect,stepsize) moves the rectangles in increments of stepsize. The lower left corner of the first rectangle is constrained to a grid of size equal to stepsize starting at the lower left corner of the figure, and all other rectangles maintain their original offset from the first rectangle.

[finalrect] = dragrect(...) returns the final positions of the rectangles when the mouse button is released. The default step size is 1.

Remarks
dragrect returns immediately if a mouse button is not currently pressed. Use dragrect in aButtonDownFcn, or from the command line in conjunction with waitforbuttonpress, to ensure that the mouse button is down when dragrect is called. dragrect returns when you release the mouse button.

If the drag ends over a figure window, the positions of the rectangles are returned in that figure's coordinate system. If the drag ends over a part of the screen not contained within a figure window, the rectangles are returned in the coordinate system of the figure over which the drag began.

Example
Drag a rectangle that is 50 pixels wide and 100 pixels in height.

```matlab
waitforbuttonpress
point1 = get(gcf,'CurrentPoint') % button down detected
rect = [point1(1,1) point1(1,2) 50 100]
[r2] = dragrect(rect)
```

See Also
rbbox, waitforbuttonpress

“Selecting Region of Interest” for related functions
Purpose
Complete pending drawing events

Syntax
drawnow

Description
drawnow flushes the event queue and updates the figure window.

Remarks
Other events that cause MATLAB to flush the event queue and draw the figure windows include

- Returning to the MATLAB prompt
- A pause statement
- A waitforbuttonpress statement
- A waitfor statement
- A getframe statement
- A figure statement

Examples
Executing the statements

```matlab
x = -pi:pi/20:pi;
plot(x,cos(x))
drawnow
title('A Short Title')
grid on
```

as an M-file updates the current figure after executing the drawnow function and after executing the final statement.

See Also
waitfor, pause, waitforbuttonpress

“Figure Windows” for related functions
Purpose
Search for nearest point

Syntax
K = dsearch(x,y,TRI,xi,yi)
K = dsearch(x,y,TRI,xi,yi,S)

Description
K = dsearch(x,y,TRI,xi,yi) returns the index into x and y of the nearest point to the point (xi,yi). dsearch requires a triangulation TRI of the points x,y obtained using delaunay. If xi and yi are vectors, K is a vector of the same size.

K = dsearch(x,y,TRI,xi,yi,S) uses the sparse matrix S instead of computing it each time:

S = sparse(TRI(:,[1 1 2 2 3 3]),TRI(:,[2 3 1 3 1 2]),1,nxy,nxy)
where nxy = prod(size(x)).

See Also
delaunay, tsearch, voronoi
**Purpose**

N-dimensional nearest point search

**Syntax**

\[
k = \text{dsearchn}(X,T,XI)
\]
\[
k = \text{dsearchn}(X,T,XI,\text{outval})
\]
\[
k = \text{dsearchn}(X,XI)
\]
\[
[k,d] = \text{dsearchn}(X,...)
\]

**Description**

\(k = \text{dsearchn}(X,T,XI)\) returns the indices \(k\) of the closest points in \(X\) 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. \(T\) is a \(numt\)-by-\(n+1\) matrix, a tessellation of the data \(X\) generated by \text{delaunayn}. The output \(k\) is a column vector of length \(p\).

\(k = \text{dsearchn}(X,T,XI,\text{outval})\) returns the indices \(k\) of the closest points in \(X\) for each point in \(XI\), unless a point is outside the convex hull. If \(XI(J,:)\) is outside the convex hull, then \(K(J)\) is assigned \(\text{outval}\), a scalar double. \(\text{Inf}\) is often used for \(\text{outval}\). If \(\text{outval}\) is [], then \(k\) is the same as in the case \(k = \text{dsearchn}(X,T,XI)\).

\(k = \text{dsearchn}(X,XI)\) performs the search without using a tessellation. With large \(X\) and small \(XI\), this approach is faster and uses much less memory.

\([k,d] = \text{dsearchn}(X,...)\) also returns the distances \(d\) to the closest points. \(d\) is a column vector of length \(p\).

**See Also**

\text{tsearch}, \text{dsearch}, \text{tsearchn}, \text{griddatan}, \text{delaunayn}
Purpose
Echo M-files during execution

Syntax

```
2echo
```

```
echo on
echo off
echo
echo fcnnname on
echo fcnnname off
echo fcnnname
echo on all
echo off all
```

Description
The `echo` command controls the echoing of M-files during execution. Normally, the commands in M-files are not displayed on the screen during execution. Command echoing is useful for debugging or for demonstrations, allowing the commands to be viewed as they execute.

The `echo` command behaves in a slightly different manner for script files and function files. For script files, the use of `echo` is simple; echoing can be either on or off, in which case any script used is affected.

```
echo on
```

Turns on the echoing of commands in all script files

```
echo off
```

Turns off the echoing of commands in all script files

```
echo
```

Toggles the echo state

With function files, the use of `echo` is more complicated. If `echo` is enabled on a function file, the file is interpreted, rather than compiled. Each input line is then displayed as it is executed. Since this results in inefficient execution, use `echo` only for debugging.

```
echo fcnnname on
```

Turns on echoing of the named function file

```
echo fcnnname off
```

Turns off echoing of the named function file

```
echo fcnnname
```

Toggles the echo state of the named function file

```
echo on all
```

Sets echoing on for all function files

```
echo off all
```

Sets echoing off for all function files
See Also

function
Purpose

Edit or create M-file

Graphical Interface

As an alternative to the edit function, select New or Open from the File menu in the MATLAB desktop or any desktop tool.

Syntax

edit
edit fun.m
edit file.ext
edit fun1 fun2 fun3 ...
edit class/fun
edit private/fun
edit class/private/fun

Description

edit opens a new editor window.
edit fun.m opens the M-file fun.m in the default editor. Note that fun.m can be a MATLAB partialpath or a complete path. If fun.m does not exist, a prompt appears asking if you want to create a new file titled fun.m. After you click Yes, the Editor/Debugger creates a blank file titled fun.m. If you do not want the prompt to appear in this situation, select that check box in the prompt. Then when you type edit fun.m, where fun.m did not previously exist, a new file called fun.m is automatically opened in the Editor. To make the prompt appear, specify it in preferences for Prompt.
edit file.ext opens the specified file.
edit fun1 fun2 fun3 ... opens fun1.m, fun2.m, fun3.m, and so on, in the default editor.
edit class/fun, edit private/fun, or edit class/private/fun can be used to edit a method, private function, or private method (for the class named class).

Remarks

To specify the default editor for MATLAB, select Preferences from the File menu. On the Editor/Debugger panel, select MATLAB editor or specify another.
UNIX Users

If you run MATLAB with the -nodisplay startup option, or run without the DISPLAY environment variable set, edit uses the External Editor command. It does not use the MATLAB Editor/Debugger, but instead uses the default editor defined for your system in $matlabroot/X11/app-defaults/Matlab.

You can specify the editor that the edit function uses or specify editor options by adding the following line to your own .Xdefaults file, located in ~home

```
matlab*externalEditorCommand: $EDITOR -option $FILE
```

where

- $EDITOR is the name of your default editor, for example, emacs; leaving it as $EDITOR means your default system editor will be used.
- -option is a valid option flag you can include for the specified editor.
- $FILE means the filename you type with the edit command will open in the specified editor.

For example,

```
emacs $FILE
```

means that when you type edit foo, the file foo will open in the emacs editor.

After adding the line to your .Xdefaults file, you must run the following before starting MATLAB:

```
xrdb -merge ~home/.Xdefaults
```

See Also

open, type
**Purpose**
Find eigenvalues and eigenvectors

**Syntax**
- \( d = \text{eig}(A) \)
- \( d = \text{eig}(A,B) \)
- \([V,D] = \text{eig}(A)\)
- \([V,D] = \text{eig}(A, 'nobalance')\)
- \([V,D] = \text{eig}(A,B)\)
- \([V,D] = \text{eig}(A,B,flag)\)

**Description**
- \( d = \text{eig}(A) \) returns a vector of the eigenvalues of matrix \( A \).
- \( d = \text{eig}(A,B) \) returns a vector containing the generalized eigenvalues, if \( A \) and \( B \) are square matrices.

**Note** If \( S \) is sparse and symmetric, you can use \( d = \text{eig}(S) \) to returns the eigenvalues of \( S \). To request eigenvectors, and in all other cases, use \( \text{eigs} \) to find the eigenvalues or eigenvectors of sparse matrices.

\([V,D] = \text{eig}(A)\) produces matrices of eigenvalues \( (D) \) and eigenvectors \( (V) \) of matrix \( A \), so that \( A*V = V*D \). Matrix \( D \) is the canonical form of \( A \)—a diagonal matrix with \( A \)'s eigenvalues on the main diagonal. Matrix \( V \) is the modal matrix—its columns are the eigenvectors of \( A \).

If \( W \) is a matrix such that \( W'*A = D*W' \), the columns of \( W \) are the left eigenvectors of \( A \). Use \( [W,D] = \text{eig}(A,') \); \( W = \text{conj}(W) \) to compute the left eigenvectors.

\([V,D] = \text{eig}(A, 'nobalance')\) finds eigenvalues and eigenvectors without a preliminary balancing step. Ordinarily, balancing improves the conditioning of the input matrix, enabling more accurate computation of the eigenvectors and eigenvalues. However, if a matrix contains small elements that are really due to roundoff error, balancing may scale them up to make them as significant as the other elements of the original matrix, leading to incorrect eigenvectors. Use the nobalance option in this event. See the balance function for more details.

\([V,D] = \text{eig}(A,B)\) produces a diagonal matrix \( D \) of generalized eigenvalues and a full matrix \( V \) whose columns are the corresponding eigenvectors so that \( A*V = B*V*D \).
[V,D] = eig(A,B,flag) specifies the algorithm used to compute eigenvalues and eigenvectors. flag can be:

- 'chol' Computes the generalized eigenvalues of A and B using the Cholesky factorization of B. This is the default for symmetric (Hermitian) A and symmetric (Hermitian) positive definite B.
- 'qz' Ignores the symmetry, if any, and uses the QZ algorithm as it would for nonsymmetric (non-Hermitian) A and B.

**Note** For eig(A), the eigenvectors are scaled so that the norm of each is 1.0. For eig(A,B), eig(A,'nobalance'), and eig(A,B,flag), the eigenvectors are not normalized.

**Remarks**

The eigenvalue problem is to determine the nontrivial solutions of the equation

\[ Ax = \lambda x \]

where A is an n-by-n matrix, x is a length n column vector, and \( \lambda \) is a scalar. The n values of \( \lambda \) that satisfy the equation are the eigenvalues, and the corresponding values of x are the right eigenvectors. In MATLAB, the function `eig` solves for the eigenvalues \( \lambda \), and optionally the eigenvectors x.

The generalized eigenvalue problem is to determine the nontrivial solutions of the equation

\[ Ax = \lambda Bx \]

where both A and B are n-by-n matrices and \( \lambda \) is a scalar. The values of \( \lambda \) that satisfy the equation are the generalized eigenvalues and the corresponding values of x are the generalized right eigenvectors.

If B is nonsingular, the problem could be solved by reducing it to a standard eigenvalue problem

\[ B^{-1}Ax = \lambda x \]

Because B can be singular, an alternative algorithm, called the QZ method, is necessary.
When a matrix has no repeated eigenvalues, the eigenvectors are always independent and the eigenvector matrix $V$ diagonalizes the original matrix $A$ if applied as a similarity transformation. However, if a matrix has repeated eigenvalues, it is not similar to a diagonal matrix unless it has a full (independent) set of eigenvectors. If the eigenvectors are not independent then the original matrix is said to be defective. Even if a matrix is defective, the solution from `eig` satisfies $A*X = X*D$.

**Examples**

The matrix

$$B = \begin{bmatrix} 3 & -2 & -.9 & 2*\text{eps} \\ -2 & 4 & 1 & -\text{eps} \\ -\text{eps}/4 & \text{eps}/2 & -1 & 0 \\ -.5 & -.5 & .1 & 1 \end{bmatrix};$$

has elements on the order of roundoff error. It is an example for which the `nobalance` option is necessary to compute the eigenvectors correctly. Try the statements

```
[VB,DB] = eig(B)
B*VB - VB*DB
[VN,DN] = eig(B,'nobalance')
B*VN - VN*DN
```

**Algorithm**

**Inputs of Type Double**

For inputs of type `double`, MATLAB uses the following LAPACK routines to compute eigenvalues and eigenvectors.

<table>
<thead>
<tr>
<th>Case</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real symmetric $A$</td>
<td>DSYEV</td>
</tr>
<tr>
<td>Real nonsymmetric $A$:</td>
<td></td>
</tr>
<tr>
<td>• With preliminary balance step</td>
<td>DGEEV (with SCLFAC = 2 instead of 8 in DGEBAL)</td>
</tr>
<tr>
<td>• $d = \text{eig}(A,'nobalance')$</td>
<td>DGEHRD, DHSEQR</td>
</tr>
<tr>
<td>• $[V,D] = \text{eig}(A,'nobalance')$</td>
<td>DGEHRD, DORGHR, DHSEQR, DTREVC</td>
</tr>
<tr>
<td>Hermitian $A$</td>
<td>ZHEEV</td>
</tr>
</tbody>
</table>
Inputs of Type Single

For inputs of type single, MATLAB uses the following LAPACK routines to compute eigenvalues and eigenvectors.

<table>
<thead>
<tr>
<th>Case</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Hermitian A:</td>
<td></td>
</tr>
<tr>
<td>• With preliminary balance step</td>
<td>ZGEEV (with SCLFAC = 2 instead of 8 in ZGEBAL)</td>
</tr>
<tr>
<td>• ( d = \text{eig}(A, 'nobalance') )</td>
<td>ZGEHRD, ZHSEQR</td>
</tr>
<tr>
<td>• ([V,D] = \text{eig}(A, 'nobalance'))</td>
<td>ZGEHRD, ZUNGHR, ZHSEQR, ZTREVC</td>
</tr>
<tr>
<td>Real symmetric A, symmetric positive definite B.</td>
<td>DSYGV</td>
</tr>
<tr>
<td>Special case:</td>
<td>DGEV</td>
</tr>
<tr>
<td>( \text{eig}(A,B, 'qz') ) for real A, B</td>
<td>ZHEGV</td>
</tr>
<tr>
<td>(same as real nonsymmetric A, real general B)</td>
<td>ZGGEV</td>
</tr>
<tr>
<td>Real nonsymmetric A, real general B</td>
<td>DGEV</td>
</tr>
<tr>
<td>Complex Hermitian A, Hermitian positive definite B.</td>
<td>ZHEGV</td>
</tr>
<tr>
<td>Special case:</td>
<td>ZGGEV</td>
</tr>
<tr>
<td>( \text{eig}(A,B, 'qz') ) for complex A or B</td>
<td>ZGGEV</td>
</tr>
<tr>
<td>(same as complex non-Hermitian A, complex B)</td>
<td></td>
</tr>
<tr>
<td>Complex non-Hermitian A, complex B</td>
<td>ZGGEV</td>
</tr>
</tbody>
</table>
See Also
balance, condeig, eigs, hess, qz, schur

References
Purpose
Find a few eigenvalues and eigenvectors of a square large sparse matrix

Syntax
\[
\begin{align*}
d &= \text{eigs}(A) \\
d &= \text{eigs}(A,B) \\
d &= \text{eigs}(A,k) \\
d &= \text{eigs}(A,B,k) \\
d &= \text{eigs}(A,k,\text{sigma}) \\
d &= \text{eigs}(A,B,k,\text{sigma}) \\
d &= \text{eigs}(A,k,\text{sigma},\text{options}) \\
d &= \text{eigs}(A,B,k,\text{sigma},\text{options}) \\
d &= \text{eigs}(\text{Afun},n) \\
d &= \text{eigs}(\text{Afun},n,B) \\
d &= \text{eigs}(\text{Afun},n,k) \\
d &= \text{eigs}(\text{Afun},n,B,k) \\
d &= \text{eigs}(\text{Afun},n,k,\text{sigma}) \\
d &= \text{eigs}(\text{Afun},n,B,k,\text{sigma}) \\
d &= \text{eigs}(\text{Afun},n,k,\text{sigma},\text{options}) \\
d &= \text{eigs}(\text{Afun},n,B,k,\text{sigma},\text{options}) \\
d &= \text{eigs}(\text{Afun},n,k,\text{sigma},\text{options},p1,p2...) \\
d &= \text{eigs}(\text{Afun},n,B,k,\text{sigma},\text{options},p1,p2...) \\
[V,D] &= \text{eigs}(A,...) \\
[V,D] &= \text{eigs}(\text{Afun},n,...) \\
[V,D,\text{flag}] &= \text{eigs}(A,...) \\
[V,D,\text{flag}] &= \text{eigs}(\text{Afun},n,...)
\end{align*}
\]

Description
\[
d = \text{eigs}(A) \text{ returns a vector of A's six largest magnitude eigenvalues.}
\]
\[
[V,D] = \text{eigs}(A) \text{ returns a diagonal matrix D of A's six largest magnitude eigenvalues and a matrix V whose columns are the corresponding eigenvectors.}
\]
\[
[V,D,\text{flag}] = \text{eigs}(A) \text{ also returns a convergence flag. If flag is 0 then all the eigenvalues converged; otherwise not all converged.}
\]
\[
\text{eigs}(A,B) \text{ solves the generalized eigenvalue problem } A*V = B*V*D. \text{ B must be symmetric (or Hermitian) positive definite and the same size as A.}
\]
\[
\text{eigs}(A,[],...) \text{ indicates the standard eigenvalue problem } A*V = V*D.
\]
\[
\text{eigs}(A,k) \text{ and eigs}(A,B,k) \text{ return the k largest magnitude eigenvalues.}
\]
eigs(A,k,sigma) and eigs(A,B,k,sigma) return k eigenvalues based on sigma, which can take any of the following values:

- **scalar** (real or complex, including 0): The eigenvalues closest to sigma. If A is a function, Afun must return \( Y = (A - sigma \cdot B)x \) (i.e., \( Y = A \backslash x \) when \( sigma = 0 \)). Note, B need only be symmetric (Hermitian) positive semi-definite.
- **'lm'**: Largest magnitude (default).
- **'sm'**: Smallest magnitude. Same as \( sigma = 0 \). If A is a function, Afun must return \( Y = A \backslash x \). Note, B need only be symmetric (Hermitian) positive semi-definite.

For real symmetric problems, the following are also options:
- **'la'**: Largest algebraic ('lr' in MATLAB 5)
- **'sa'**: Smallest algebraic ('sr' in MATLAB 5)
- **'be'**: Both ends (one more from high end if \( k \) is odd)

For nonsymmetric and complex problems, the following are also options:
- **'lr'**: Largest real part
- **'sr'**: Smallest real part
- **'li'**: Largest imaginary part
- **'si'**: Smallest imaginary part

**Note** The MATLAB 5 value \( sigma = 'be' \) is obsolete for nonsymmetric and complex problems.
eigs(A,K,sigma,opts) and eigs(A,B,k,sigma,opts) specify an options structure. Default values are shown in brackets ({}).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>options.issym</td>
<td>1 if A or A-sigma*B represented by Afun is symmetric, 0 otherwise.</td>
<td>[{0}</td>
</tr>
<tr>
<td>options.isreal</td>
<td>1 if A or A-sigma*B represented by Afun is real, 0 otherwise.</td>
<td>[0</td>
</tr>
<tr>
<td>options.tol</td>
<td>Convergence: Ritz estimate residual &lt;= tol*norm(A).</td>
<td>[scalar</td>
</tr>
<tr>
<td>options.maxit</td>
<td>Maximum number of iterations.</td>
<td>[integer</td>
</tr>
<tr>
<td>options.p</td>
<td>Number of basis vectors. p &gt;= 2k (p &gt;= 2k+1 real nonsymmetric) advised. Note: p must satisfy k &lt; p &lt;= n for real symmetric, k+1 &lt; p &lt;= n otherwise.</td>
<td>[integer</td>
</tr>
<tr>
<td>options.v0</td>
<td>Starting vector.</td>
<td>Randomly generated by ARPACK</td>
</tr>
<tr>
<td>options.disp</td>
<td>Diagnostic information display level.</td>
<td>[0</td>
</tr>
<tr>
<td>options.cholB</td>
<td>1 if B is really its Cholesky factor chol(B), 0 otherwise.</td>
<td>[{0}</td>
</tr>
<tr>
<td>options.permB</td>
<td>Permutation vector permB if sparse B is really chol(B(permB,permB)).</td>
<td>[permB</td>
</tr>
</tbody>
</table>

**Note** MATLAB 5 options stagtol and cheb are no longer allowed.
eigs(Afun,n,...) accepts the function Afun instead of the matrix A.
y = Afun(x) should return:

- \( A^*x \) if \( sigma \) is not specified, or is a string other than 'sm'
- \( A\backslash x \) if \( sigma \) is 0 or 'sm'
- \( (A-sigma*I)\backslash x \) if \( sigma \) is a nonzero scalar (standard eigenvalue problem). \( I \) is an identity matrix of the same size as \( A \).
- \( (A-sigma*B)\backslash x \) if \( sigma \) is a nonzero scalar (generalized eigenvalue problem)

\( n \) is the size of \( A \). The matrix \( A, A-sigma*I \) or \( A-sigma*B \) represented by Afun is assumed to be real and nonsymmetric unless specified otherwise by opts.isreal and opts.issym. In all the eigs syntaxes, eigs(A,...) can be replaced by eigs(Afun,n,...).

eigs(Afun,n,k,sigma,opts,p1,p2,...) and eigs(Afun,n,B,k,sigma,opts,p1,p2,...) provide for additional arguments which are passed to Afun(x,p1,p2,...).

**Remarks**

d = eigs(A,k) is not a substitute for

- \( d = \text{eig}(\text{full}(A)) \)
- \( d = \text{sort}(d) \)
- \( d = d(end-k+1:end) \)

but is most appropriate for large sparse matrices. If the problem fits into memory, it may be quicker to use eigs(Afun,n,...).

**Algorithm**
eigs provides the reverse communication required by the Fortran library ARPACK, namely the routines DSAUPD, DSEUPD, DNAUPD, DNEUPD, ZNAUPD, and ZNEUPD.

**Examples**

**Example 1:** This example shows the use of function handles.

\[
A = \text{delsq}(\text{numgrid}('C',15));
\]
\[
d1 = \text{eigs}(A,5,'sm');
\]

Equivalently, if \( dnRk \) is the following one-line function:

\[
\text{function } y = dnRk(x,R,k)
\]
\[ y = (\text{delsq(numgrid}(R,k))) \backslash x; \]

then pass \( \text{dnRk} \)’s additional arguments, ’C’ and 15, to \text{eigs}.

\[
\begin{align*}
n &= \text{size}(A,1); \\
opts.\text{issym} &= 1; \\
d2 &= \text{eigs}(@\text{dnRk}, n, 5, 'sm', \text{opts}, 'C', 15);
\end{align*}
\]

**Example 2:** \text{west0479} is a real 479-by-479 sparse matrix with both real and pairs of complex conjugate eigenvalues. \text{eig} computes all 479 eigenvalues. \text{eigs} easily picks out the largest magnitude eigenvalues.

This plot shows the 8 largest magnitude eigenvalues of \text{west0479} as computed by \text{eig} and \text{eigs}.

\[
\begin{align*}
\text{load west0479} \\
d &= \text{eig}(\text{full}(\text{west0479})) \\
dlm &= \text{eigs}(\text{west0479}, 8) \\
[\text{dum}, \text{ind}] &= \text{sort}(\text{abs}(d)); \\
\text{plot}(dlm, 'k+'); \\
\text{hold on} \\
\text{plot}(d(\text{ind}\text{(end-7:end)}), 'ks') \\
\text{hold off} \\
\text{legend}('\text{eigs}(\text{west0479}, 8)', '\text{eig}(\text{full}(\text{west0479}))')
\end{align*}
\]
Example 3: \( A = \text{delsq}(\text{numgrid}(\text{'C'},30)) \) is a symmetric positive definite matrix of size 632 with eigenvalues reasonably well-distributed in the interval \((0, 8)\), but with 18 eigenvalues repeated at 4. The \text{eig} function computes all 632 eigenvalues. It computes and plots the six largest and smallest magnitude eigenvalues of \( A \) successfully with:

\[
\begin{align*}
A &= \text{delsq}(\text{numgrid}(\text{'C'},30)); \\
d &= \text{eig}(\text{full}(A)); \\
[dum, ind] &= \text{sort}(\text{abs}(d)); \\
dlm &= \text{eigs}(A); \\
dsm &= \text{eigs}(A, 6, 'sm'); \\
\text{subplot}(2,1,1) \\
\text{plot}(dlm, 'k+') \\
\text{hold on} \\
\text{plot}(d(ind(end:-1:end-5)), 'ks') \\
\text{hold off} \\
\text{legend}('\text{eigs}(A)', '\text{eig}(\text{full}(A))', 3) \\
\text{set(gca, 'XLim', [0.5 6.5])}
\end{align*}
\]
However, the repeated eigenvalue at 4 must be handled more carefully. The call `eigs(A,18,4.0)` to compute 18 eigenvalues near 4.0 tries to find eigenvalues of \( A - 4.0 \cdot I \). This involves divisions of the form \( 1/(\text{lambda} - 4.0) \), where `lambda` is an estimate of an eigenvalue of \( A \). As `lambda` gets closer to 4.0, `eigs` fails. We must use `sigma` near but not equal to 4 to find those 18 eigenvalues.

\[
\text{sigma} = 4 - 1e^{-6}
\]

\[
[V,D] = \text{eigs}(A,18,\text{sigma})
\]
The plot shows the 20 eigenvalues closest to 4 that were computed by `eig`, along with the 18 eigenvalues closest to $4 - 1 \times 10^{-6}$ that were computed by `eigs`.

**See Also**

`arpackc`, `eig`, `svds`

**References**


Purpose

Jacobi elliptic functions

Syntax

\[ \text{[SN,CN,DN]} = \text{ellipj}(U,M) \]
\[ \text{[SN,CN,DN]} = \text{ellipj}(U,M,tol) \]

Definition

The Jacobi elliptic functions are defined in terms of the integral:

\[ u = \int_{0}^{\phi} \frac{d\theta}{(1 - m \sin^2 \theta)^{\frac{1}{2}}} \]

Then

\[ sn(u) = \sin \phi, \ cn(u) = \cos \phi, \ dn(u) = (1 - m \sin^2 \phi)^{\frac{1}{2}}, \ am(u) = \phi \]

Some definitions of the elliptic functions use the modulus \( k \) instead of the parameter \( m \). They are related by

\[ k^2 = m = \sin^2 \alpha \]

The Jacobi elliptic functions obey many mathematical identities; for a good sample, see [1].

Description

\[ \text{[SN,CN,DN]} = \text{ellipj}(U,M) \] returns the Jacobi elliptic functions SN, CN, and DN, evaluated for corresponding elements of argument \( U \) and parameter \( M \). Inputs \( U \) and \( M \) must be the same size (or either can be scalar).

\[ \text{[SN,CN,DN]} = \text{ellipj}(U,M,tol) \] computes the Jacobi elliptic functions to accuracy \( tol \). The default is \( \text{eps} \); increase this for a less accurate but more quickly computed answer.

Algorithm

\text{ellipj} computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean [1]. It starts with the triplet of numbers:

\[ a_0 = 1, \ b_0 = (1 - m)^{\frac{1}{2}}, \ c_0 = (m)^{\frac{1}{2}} \]
ellipj computes successive iterates with

\[ a_i = \frac{1}{2}(a_{i-1} + b_{i-1}) \]
\[ b_i = (a_{i-1} b_{i-1})^{\frac{1}{2}} \]
\[ c_i = \frac{1}{2}(a_{i-1} - b_{i-1}) \]

Next, it calculates the amplitudes in radians using:

\[ \sin(2\phi_{n-1} - \phi_n) = \frac{c_n}{a_n} \sin(\phi_n) \]

being careful to unwrap the phases correctly. The Jacobian elliptic functions are then simply:

\[ sn(u) = \sin\phi_0 \]
\[ cn(u) = \cos\phi_0 \]
\[ dn(u) = (1 - m \cdot sn(u)^2)^{\frac{1}{2}} \]

**Limitations**

The `ellipj` function is limited to the input domain \( 0 \leq m \leq 1 \). Map other values of \( m \) into this range using the transformations described in [1], equations 16.10 and 16.11. \( u \) is limited to real values.

**See Also**

`ellipke`

**References**

Purpose
Complete elliptic integrals of the first and second kind

Syntax
K = ellipke(M)
[K,E] = ellipke(M)
[K,E] = ellipke(M,tol)

Definition
The complete elliptic integral of the first kind [1] is

\[ K(m) = F(\pi/2|m) \]

where \( F \), the elliptic integral of the first kind, is

\[ K(m) = \int_0^1 \frac{dt}{\sqrt{(1-t^2)(1-mt^2)}} = \int_0^{\pi/2} (1-m \sin^2 \theta)^{-1/2} d\theta \]

The complete elliptic integral of the second kind

\[ E(m) = E(K(m)) = E(\pi/2|m) \]

is

\[ E(m) = \int_0^1 (1-t^2)^{-1/2}(1-mt^2)^{1/2} dt = \int_0^{\pi/2} (1-m \sin^2 \theta)^{1/2} d\theta \]

Some definitions of \( K \) and \( E \) use the modulus \( k \) instead of the parameter \( m \).
They are related by

\[ k^2 = m = \sin^2 \alpha \]

Description
\( K = \text{ellipke}(M) \) returns the complete elliptic integral of the first kind for the elements of \( M \).
\n\([K,E] = \text{ellipke}(M)\) returns the complete elliptic integral of the first and second kinds.

\([K,E] = \text{ellipke}(M,\text{tol})\) computes the complete elliptic integral to accuracy \( \text{tol} \). The default is \( \text{eps} \); increase this for a less accurate but more quickly computed answer.
ellipke

Algorithm ellipke computes the complete elliptic integral using the method of the arithmetic-geometric mean described in [1], section 17.6. It starts with the triplet of numbers

\[ a_0 = 1, \quad b_0 = (1 - m)^{\frac{1}{2}}, \quad c_0 = (m)^{\frac{1}{2}} \]

ellipke computes successive iterations of \( a_i, b_i, \) and \( c_i \) with

\[ a_i = \frac{1}{2}(a_{i-1} + b_{i-1}) \]
\[ b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}} \]
\[ c_i = \frac{1}{2}(a_{i-1} - b_{i-1}) \]

stopping at iteration \( n \) when \( cn = 0 \), within the tolerance specified by \( \text{eps} \). The complete elliptic integral of the first kind is then

\[ K(m) = \frac{\pi}{2a_n} \]

Limitations ellipke is limited to the input domain \( 0 \leq m \leq 1 \).

See Also ellipj

Purpose
Generate ellipsoid

Syntax
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)
ellipsoid(axes_handle,...)
ellipsoid(...)

Description
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n) generates three n+1-by-n+1 matrices so that surf(x,y,z) produces an ellipsoid with center (xc,yc,zc) and radii (xr,yr,zr).

[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr) uses n = 20.
ellipsoid(axes_handle,...) plots into the axes with handle axes_handle instead of the current axes (gca).
ellipsoid(...) with no output arguments graphs the ellipsoid as a surface.

Algorithm
ellipsoid generates the data using the following equation:

\[
\frac{(x-x_c)^2}{x_r^2} + \frac{(y-y_c)^2}{y_r^2} + \frac{(z-z_c)^2}{z_r^2}
\]

See Also
cylinder, sphere, surf

"Polygons and Surfaces" for related functions

\[
Y = \sin(X);
E = \text{std}(Y) \times \text{ones(size}(X)));
\]
else

Purpose  Conditionally execute statements

Syntax
if expression
    statements1
else
    statements2
end

Description  else is used to delineate an alternate block of statements. If expression evaluates as false, MATLAB executes the one or more commands denoted here as statements2.

A true expression has either a logical true or nonzero value. For nonscalar expressions, (for example, “if (matrix A is less than matrix B)”), true means that every element of the resulting matrix has a logical true or nonzero value.

Expressions usually involve relational operations such as (count < limit) or isreal(A). Simple expressions can be combined by logical operators (&, |, ~) into compound expressions such as (count < limit) & ((height - offset) >= 0).

See if for more information.

Examples  In this example, if both of the conditions are not satisfied, then the student fails the course.

    if ((attendance >= 0.90) & (grade_average >= 60))
        pass = 1;
    else
        fail = 1;
    end;

See Also  if, elseif, end, for, while, switch, break, return, relational operators, logical operators (elementwise and short-circuit)
**Purpose**

Conditionally execute statements

**Syntax**

```
if expression1
    statements1
elseif expression2
    statements2
end
```

**Description**

If `expression1` evaluates as false and `expression2` as true, MATLAB executes the one or more commands denoted here as `statements2`.

A true expression has either a logical true or nonzero value. For nonscalar expressions, (for example, is matrix A less then matrix B), true means that every element of the resulting matrix has a logical true or nonzero value.

Expressions usually involve relational operations such as `(count < limit)` or `isreal(A)`. Simple expressions can be combined by logical operators (`&`, `|`, `~`) into compound expressions such as `(count < limit) & ((height - offset) >= 0)`.

See if for more information.

**Remarks**

else if, with a space between the else and the if, differs from elseif, with no space. The former introduces a new, nested if, which must have a matching end. The latter is used in a linear sequence of conditional statements with only one terminating end.

The two segments shown below produce identical results. Exactly one of the four assignments to `x` is executed, depending upon the values of the three logical expressions, A, B, and C.

```matlab
if A
    x = a
else
    if B
        x = b
    else
        if C
            x = c
        else
            x = d
        end
    end
end
```

```matlab
if A
    x = a
elseif B
    x = b
elseif C
    x = c
else
    x = d
end
```
elseif

end
end

Examples

Here is an example showing if, else, and elseif.

```matlab
for m = 1:k
    for n = 1:k
        if m == n
            a(m,n) = 2;
        elseif abs(m-n) == 2
            a(m,n) = 1;
        else
            a(m,n) = 0;
        end
    end
end
```

For k=5 you get the matrix

```
a =

2  0  1  0  0
0  2  0  1  0
1  0  2  0  1
0  1  0  2  0
0  0  1  0  2
```

See Also

if, else, end, for, while, switch, break, return, relational operators, logical operators (elementwise and short-circuit)
Purpose
Terminate for, while, switch, try, and if statements or indicate last index

Syntax
while expression % (or if, for, or try)
    statements
end
B = A(index:end,index)

Description
end is used to terminate for, while, switch, try, and if statements. Without an end statement, for, while, switch, try, and if wait for further input. Each end is paired with the closest previous unpaired for, while, switch, try, or if and serves to delimit its scope.

The end command also serves as the last index in an indexing expression. In that context, end = (size(x,k)) when used as part of the kth index. Examples of this use are X(3:end) and X(1,1:2:end-1). When using end to grow an array, as in X(end+1)=5, make sure X exists first.

You can overload the end statement for a user object by defining an end method for the object. The end method should have the calling sequence end(obj,k,n), where obj is the user object, k is the index in the expression where the end syntax is used, and n is the total number of indices in the expression. For example, consider the expression

\[ A(\text{end-1},:) \]

MATLAB will call the end method defined for A using the syntax

\[ \text{end}(A,1,2) \]

Examples
This example shows end used with the for and if statements.

\[
\text{for } k = 1:n \\
\quad \text{if } a(k) == 0 \\
\qquad a(k) = a(k) + 2; \\
\quad \text{end} \\
\text{end}
\]

In this example, end is used in an indexing expression.

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

\[ A = \]
end

\[
B = A(\text{end},2:\text{end})
\]

\[
B = \\
18 \quad 25 \quad 2 \quad 9
\]

**See Also** break, for, if, return, switch, try, while
Purpose
End of month

Syntax
E = eomday(Y,M)

Description
E = eomday(Y,M) returns the last day of the year and month given by corresponding elements of arrays Y and M.

Examples
Because 1996 is a leap year, the statement eomday(1996,2) returns 29.
To show all the leap years in this century, try:

```matlab
y = 1900:1999;
E = eomday(y,2*ones(length(y),1));
y(find(E==29))'
```

```plaintext
ans =
Columns 1 through 6
  1904  1908  1912  1916  1920  1924
Columns 7 through 12
  1928  1932  1936  1940  1944  1948
Columns 13 through 18
Columns 19 through 24
```

See Also
datenum, datevec, weekday
**Purpose**
Floating-point relative accuracy

**Syntax**
```
esps
   d = eps(X)
esps('double')
esps('single')
```

**Description**
esps returns the distance from 1.0 to the next largest double-precision number, that is \( \text{eps} = 2^{-52} \).

\( d = \text{eps}(X) \) is the positive distance from \(|X|\) to the next larger in magnitude floating point number of the same precision as \( X \). \( X \) may be either double precision or single precision. For all \( X \),

\[
\text{eps}(X) = \text{eps}(-X) = \text{eps}(abs(X))
\]

\text{eps('double')} is the same as \text{eps} or \text{eps}(1.0).
\text{eps('single')} is the same as \text{eps(single(1.0))} or \text{single(2^{-23})}.

Except for denormals, if \( 2^E \leq \text{abs}(X) < 2^{(E+1)} \), then

\[
\text{eps}(X) = 2^{(E-23)} \text{ if } \text{isa}(X, \text{'single'}) \n\]
\[
\text{eps}(X) = 2^{(E-52)} \text{ if } \text{isa}(X, \text{'double'}) \n\]

Replace expressions of the form

\[
\text{if } Y < \text{eps} \times \text{ABS}(X) \n\]
with

\[
\text{if } Y < \text{eps}(X) \n\]

**Examples**
do the precision
\[
\text{eps}(1/2) = 2^{(-53)}
\]
\[
\text{eps}(1) = 2^{(-52)}
\]
\[
\text{eps}(2) = 2^{(-51)}
\]
\[
\text{eps}(\text{realmax}) = 2^{971}
\]
\[
\text{eps}(0) = 2^{(-1074)}
\]
\[
\text{if}(|x|) \leq \text{realmin}, \text{eps}(x) = 2^{(-1074)}
\]
\[
\text{eps}(\text{Inf}) = \text{NaN}
\]
\[
\text{eps}(\text{NaN}) = \text{NaN}
\]
single precision

2-670
\[ \text{eps}(\text{single}(1/2)) = 2^{-24} \]
\[ \text{eps}(\text{single}(1)) = 2^{-23} \]
\[ \text{eps}(\text{single}(2)) = 2^{-22} \]
\[ \text{eps}(\text{realmax('single'))} = 2^{104} \]
\[ \text{eps}(\text{single}(0)) = 2^{-149} \]
\[ \text{if}(\text{abs}(x) <= \text{realmin('single'))}, \text{eps}(x) = 2^{-149} \]
\[ \text{eps}(\text{single(Inf))} = \text{single(NaN)} \]
\[ \text{eps}(\text{single(NaN))} = \text{single(NaN)} \]

See Also: \text{realmax, realmin}
erf, erfc, erfcx, erfinv, erfcinv

**Purpose**  
Error functions

**Syntax**
- \( Y = \text{erf}(X) \)  
  Error function
- \( Y = \text{erfc}(X) \)  
  Complementary error function
- \( Y = \text{erfcx}(X) \)  
  Scaled complementary error function
- \( X = \text{erfinv}(Y) \)  
  Inverse error function
- \( X = \text{erfcinv}(Y) \)  
  Inverse complementary error function

**Definition**
The error function \( \text{erf}(x) \) is twice the integral of the Gaussian distribution with 0 mean and variance of \( 1/2 \).

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt
\]

The complementary error function \( \text{erfc}(x) \) is defined as

\[
\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1 - \text{erf}(x)
\]

The scaled complementary error function \( \text{erfcx}(x) \) is defined as

\[
\text{erfcx}(x) = e^{x^2} \text{erfc}(x)
\]

For large \( x \), \( \text{erfcx}(x) \) is approximately \( \left( \frac{1}{\sqrt{\pi}x} \right) \frac{1}{x} \)

**Description**
- \( Y = \text{erf}(X) \) returns the value of the error function for each element of real array \( X \).
- \( Y = \text{erfc}(X) \) computes the value of the complementary error function.
- \( Y = \text{erfcx}(X) \) computes the value of the scaled complementary error function.
- \( X = \text{erfinv}(Y) \) returns the value of the inverse error function for each element of \( Y \). Elements of \( Y \) must be in the interval \([-1, 1]\). The function \( \text{erfinv} \) satisfies \( y = \text{erf}(x) \) for \(-1 \leq y \leq 1 \) and \(-\infty \leq x \leq \infty \).
- \( X = \text{erfcinv}(Y) \) returns the value of the inverse of the complementary error function for each element of \( Y \). Elements of \( Y \) must be in the interval \([0, 2]\). The function \( \text{erfcinv} \) satisfies \( y = \text{erfc}(x) \) for \( 2 \geq y \geq 0 \) and \(-\infty \leq x \leq \infty \).
**Remarks**

The relationship between the complementary error function \( \text{erfc} \) and the standard normal probability distribution returned by the Statistics Toolbox function \( \text{normcdf} \) is

\[
\text{normcdf}(x) = 0.5 \times \text{erfc}(-x/\sqrt{2})
\]

The relationship between the inverse complementary error function \( \text{erfcinv} \) and the inverse standard normal probability distribution returned by the Statistics Toolbox function \( \text{norminv} \) is

\[
\text{norminv}(p) = -\sqrt{2} \times \text{erfcinv}(2p)
\]

**Examples**

\( \text{erfinv}(1) \) is Inf

\( \text{erfinv}(-1) \) is -Inf.

For \( \text{abs}(Y) > 1 \), \( \text{erfinv}(Y) \) is NaN.

**Algorithms**

For the error functions, the MATLAB code is a translation of a Fortran program by W. J. Cody, Argonne National Laboratory, NETLIB/SPECFUN, March 19, 1990. The main computation evaluates near-minimax rational approximations from [1].

For the inverse of the error function, rational approximations accurate to approximately six significant digits are used to generate an initial approximation, which is then improved to full accuracy by one step of Halley’s method.

**References**

error

Purpose
Display error messages

Syntax
error('message')
error('message',a1,a2, ...)
error('message_id','message')
error('message_id','message',a1,a2,...)

Description
error('message') displays an error message and returns control to the keyboard. The error message contains the input string message.
The error command has no effect if message is a null string.

error('message',a1,a2,...) displays a message string that contains formatting conversion characters, such as those used with the MATLAB sprintf function. Each conversion character in message is converted to one of the values a1, a2, ... in the argument list.

Note
MATLAB converts special characters (like \n and %d) in the error message string only when you specify more than one input argument with error. See Example 3 below.

error('message_id','message') attaches a unique message identifier, or message_id, to the error message. The identifier enables you to better identify the source of an error. See “Message Identifiers” and “Using Message Identifiers with lasterr” in the MATLAB documentation for more information on the message_id argument and how to use it.

error('message_id','message',a1,a2,...) includes formatting conversion characters in message, and the character translations a1, a2, ...

Examples
Example 1
The error function provides an error return from M-files:

function foo(x,y)
   if nargin ~= 2
      error('Wrong number of input arguments')
   end

2-674
The returned error message looks like this:

```matlab
foo(pi)

??? Error using ==> foo
Wrong number of input arguments
```

**Example 2**

Specify a message identifier and error message string with `error`:

```matlab
error('MyToolbox:angleTooLarge', ...
     'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterr` to determine the message identifier and error message string for the failing operation:

```matlab
[errmsg, msgid] = lasterr
errmsg =
   The angle specified must be less than 90 degrees.
msgid =
   MyToolbox:angleTooLarge
```

**Example 3**

MATLAB converts special characters (like `\n` and `%d`) in the error message string only when you specify more than one input argument with `error`. In the single argument case shown below, `\n` is taken to mean backslash-n. It is not converted to a newline character:

```matlab
error('In this case, the newline \n is not converted.')
??? In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This holds true regardless of whether the additional argument supplies conversion values or is a message identifier:

```matlab
error('ErrorTests:convertTest', ...
     'In this case, the newline \n is converted.')
??? In this case, the newline is converted.
```

**See Also**

`lasterr`, `lasterror`, `rethrow`, `errordlg`, `warning`, `lastwarn`, `warndlg`, `dbstop`, `disp`, `sprintf`
**Purpose**
Plot error bars along a curve

**Syntax**
- `errorbar(Y,E)`
- `errorbar(X,Y,E)`
- `errorbar(X,Y,L,U)`
- `errorbar(...,LineSpec)`
- `h = errorbar(...)`
- `errorbar('v6',...)`

**Description**
Error bars show the confidence level of data or the deviation along a curve.

`errorbar(Y,E)` plots `Y` and draws an error bar at each element of `Y`. The error bar is a distance of `E(i)` above and below the curve so that each bar is symmetric and `2*E(i)` long.

`errorbar(X,Y,E)` plots `Y` versus `X` with symmetric error bars `2*E(i)` long. `X`, `Y`, `E` must be the same size. When they are vectors, each error bar is a distance of `E(i)` above and below the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `E(i,j)` above and below the point defined by `(X(i,j),Y(i,j))`.

`errorbar(X,Y,L,U)` plots `X` versus `Y` with error bars `L(i)+U(i)` long specifying the lower and upper error bars. `X`, `Y`, `L`, and `U` must be the same size. When they are vectors, each error bar is a distance of `L(i)` below and `U(i)` above the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `L(i,j)` below and `U(i,j)` above the point defined by `(X(i,j),Y(i,j))`.

`errorbar(...,LineSpec)` draws the error bars using the line type, marker symbol, and color specified by `LineSpec`.

`h = errorbar(...)` returns handles to the `errorbar` series objects created. `errorbar` creates one object for vector input arguments and one object per column for matrix input arguments. See `errorbar` series properties for more information.
Backward Compatible Version

`hlines = errorbar('v6',...)` returns the handles of line objects instead of errorbarseries objects for compatibility with MATLAB 6.5 and earlier.

See Plot Objects and Backward Compatibility for more information.

Remarks

When the arguments are all matrices, `errorbar` draws one line per matrix column. If `X` and `Y` are vectors, they specify one curve.

Examples

Draw symmetric error bars that are two standard deviation units in length.

```matlab
X = 0:pi/10:pi;
Y = sin(X);
E = std(Y)*ones(size(X));
errorbar(X,Y,E)
```

See Also

`LineSpec`, `plot`, `std`

“Basic Plots and Graphs” for related functions

Error Bounds for related information
errorbar

See “Errorbarseries Properties” for property descriptions
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 errorbarseries objects. See Plot Objects for more information on errorbarseries 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 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 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.
**Errorbarseries 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 errorbarseries 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 errorbarseries object.* An array containing the handles of all line objects parented to the errorbarseries 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 errorbar `Children` property unless you set the Root `ShowHiddenHandles` property to `on`:

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

**Clipping** `{on} | off`

*Clipping mode.* MATLAB clips errorbar 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 errorbar lines.* A three-element RGB vector or one of the MATLAB predefined names, specifying the curve and error bar color. See the `ColorSpec` reference page for more information on specifying color.

For example, the following statement would produce an errorbar graph with both the curve and error bars colored red.

```
h = errorbar(Y,randn(10,1), 'Color', 'r');
```

**CreateFcn** string or function handle

*Not available on errorbarseries objects.*
**DeleteFcn**  
String or function handle  

*Callback executed during object deletion.* A callback that executes when the errorbarseries object is deleted (e.g., this might happen when you issue a delete command on the errorbarseries 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 errorbarseries objects appearing in these legends.

**EraseMode**  

{normal} | none | xor | background  

*Erase mode.* This property controls the technique MATLAB uses to draw and erase errorbar child objects (the lines used to construct the errorbar 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.

### 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., perform an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

Set the axes background color with the axes Color property. Set the figure background color with the figure Color property.

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

**HandleVisibility**  {on} | callback | off

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

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 errorbarseries 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 curve and error bars that compose the errorbar graph. If HitTest is off, clicking the errorbarseries object selects the object below it (which is usually the axes containing it).

**HitTestArea**
on | {off}
Select errorbarseries object on lines or area of graph. This property enables you to select errorbarseries objects in two ways:

- Select by clicking curve and error bars (default).
- Select by clicking anywhere in the extent of the errorbar graph.
When HitTestArea is off, you must click the curve or error bars to select the
errorbarseries object. When HitTestArea is on, you can select the
errorbarseries object by clicking anywhere within the extent of the errorbar
graph (i.e., anywhere within a rectangle that encloses all the lines).

**Interruptible** {on} | off

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

Only callbacks defined for the ButtonDownFcn 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 an errorbar 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.

**LData**

*Errorbar length below data point.* The errorbar function uses this data to
determine the length of the errorbar below each data point. Specify these
values in data units. See also UData.

**LDataSource**

*Link LData to MATLAB variable.* Set this property to a MATLAB variable that
is evaluated in the base workspace to generate the LData.

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

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.
**Errorbarseries Properties**

**LineStyle**

\{-\} | \|--\ | : | \equal\ | none

*Line style.* This property specifies the line style used for the curve and error bars. 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>\equal\</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>none</td>
<td>No line</td>
</tr>
</tbody>
</table>

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

**LineWidth**

Scalar

*The width of the curve and error bar 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 data points defining the curve. 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>
</tbody>
</table>
## Errorbarseries Properties

<table>
<thead>
<tr>
<th>Marker Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Square</td>
</tr>
<tr>
<td>d</td>
<td>Diamond</td>
</tr>
<tr>
<td>^</td>
<td>Upward-pointing triangle</td>
</tr>
<tr>
<td>v</td>
<td>Downward-pointing triangle</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right-pointing triangle</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left-pointing triangle</td>
</tr>
<tr>
<td>p</td>
<td>Five-pointed star (pentagram)</td>
</tr>
<tr>
<td>h</td>
<td>Six-pointed star (hexagram)</td>
</tr>
<tr>
<td>none</td>
<td>No marker (default)</td>
</tr>
</tbody>
</table>

**MarkerEdgeColor**  
ColorSpec | none | {auto}

*Marker edge color.* The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none specifies no color, which makes nonfilled markers invisible. auto sets MarkerEdgeColor to the same color as the Color property.

**MarkerFaceColor**  
ColorSpec | {none} | auto

*Marker face color.* The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). ColorSpec defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the axes color, or to the figure color if the axes Color property is set to none (which is the factory default for axes objects).

**MarkerSize**  
size in points

*Marker size.* A scalar specifying the size of the marker in points. The default value for MarkerSize is 6 points (1 point = 1/72 inch). Note that MATLAB draws the point marker (specified by the ‘.’ symbol) at one-third the specified size.
Errorbarseries Properties

**Parent**

object handle

*Parent of errorbarseries object.* This property contains the handle of the errorbarseries object’s parent. The parent of an errorbarseries 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 errorbarseries 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 curve and error bars. 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 an errorbarseries object and set the `Tag` property:

```matlab
t = errorbar(Y,E,'Tag','errorbar1')
```

When you want to access the errorbarseries object, you can use `findobj` to find the errorbarseries object’s handle.

The following statement changes the `MarkerFaceColor` property of the object whose `Tag` is `errorbar1`.

```matlab
set(findobj('Tag','errorbar1'),'MarkerFaceColor','red')
```
Errorbarseries Properties

Type  string (read only)

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

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

UData  array equal in size to XData and YData

**Errorbar length above data point.** The errorbar function uses this data to determine the length of the errorbar above each data point. Specify these values in data units.

UDatasource  string (MATLAB variable)

**Link uData to MATLAB variable.** Set this property to a MATLAB variable that 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.

UIContextMenu  handle of a uicontextmenu object

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

UserData  array

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

Visible  {on} | off

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

**XData**

array

*X-coordinates of the curve.* The errorbar function plots a curve using the *x*-axis coordinates in the XData array. XData must be the same size as YData.

If you do not specify XData (i.e., the input argument `x`), the errorbar function uses the indices of YData to create the curve. 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 errorbar function sets this property to manual.

If you set XDataMode to auto after having specified XData, the errorbar function resets the *x* tick-mark labels to the indices of the YData.

**XDataSource**

string (MATLAB variable)

*Link XData to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

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

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

See the refreshdata reference page for more information.

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

scalar, vector, or matrix

Data defining curve. YData contains the data defining the curve. If YData is a matrix, the errorbar function displays a curve with error bars for each column in the matrix.

The input argument Y in the errorbar function calling syntax assigns values to YData.

YDataSource

string (MATLAB variable)

Link YData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData.

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

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

See the refreshdata reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.
**Purpose**
Create and display an error dialog box

**Syntax**
```matlab
errordlg
errordlg('errorstring')
errordlg('errorstring','dlgname')
errordlg('errorstring','dlgname','on')
h = errordlg(...)```

**Description**
`errordlg` creates an error dialog box, or if the named dialog exists, `errordlg` pops the named dialog in front of other windows.

`errordlg` displays a dialog box named 'Error Dialog' that contains the string 'This is the default error string.'

`errordlg('errorstring')` displays a dialog box named 'Error Dialog' that contains the string 'errorstring'.

`errordlg('errorstring','dlgname')` displays a dialog box named 'dlgname' that contains the string 'errorstring'.

`errordlg('errorstring','dlgname','on')` specifies whether to replace an existing dialog box having the same name. 'on' brings an existing error dialog having the same name to the foreground. In this case, `errordlg` does not create a new dialog.

`h = errordlg(...)` returns the handle of the dialog box.

**Remarks**
MATLAB sizes the dialog box to fit the string 'errorstring'. The error dialog box has an OK pushbutton and remains on the screen until you press the OK button or the Return key. After pressing the button, the error dialog box disappears.

The appearance of the dialog box depends on the windowing system you use.

**Examples**
The function
```matlab
erroldlg('File not found','File Error');```
errordlg

displays this dialog box:

![File Error](Image)

See Also

dialog, helpdlg, msgbox, questdlg, warndlg

“Predefined Dialog Boxes” for related functions
etime

Purpose
Elapsed time

Syntax
e = etime(t2,t1)

Description
e = etime(t2,t1) returns the time in seconds between vectors t1 and t2. The
   two vectors must be six elements long, in the format returned by clock:

   T = [Year Month Day Hour Minute Second]

Examples
Calculate how long a 2048-point real FFT takes.

   x = rand(2048,1);
   t = clock; fft(x); etime(clock,t)
   ans =
       0.4167

Limitations
As currently implemented, the etime function fails across month and year
   boundaries. Since etime is an M-file, you can modify the code to work across
   these boundaries if needed.

See Also
clock, cputime, tic, toc
etree

Purpose

Elimination tree

Syntax

\[
\begin{align*}
    p &= \text{etree}(A) \\
    p &= \text{etree}(A, 'col') \\
    p &= \text{etree}(A, 'sym') \\
    [p, q] &= \text{etree}(\ldots)
\end{align*}
\]

Description

\( p = \text{etree}(A) \) returns an elimination tree for the square symmetric matrix
whose upper triangle is that of \( A \). \( p(j) \) is the parent of column \( j \) in the tree, or
0 if \( j \) is a root.

\( p = \text{etree}(A, 'col') \) returns the elimination tree of \( A' * A \).

\( p = \text{etree}(A, 'sym') \) is the same as \( p = \text{etree}(A) \).

\([p, q] = \text{etree}(\ldots)\) also returns a postorder permutation \( q \) of the tree.

See Also

\text{treelayout, treeplot, etreeplot}
etreeplot

**Purpose**
Plot elimination tree

**Syntax**
- etreeplot(A)
- etreeplot(A,nodeSpec,edgeSpec)

**Description**
etreeplot(A) plots the elimination tree of A (or A+A', if non-symmetric).
etreeplot(A,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, treeplot, treelayout
**Purpose**
Execute a string containing a MATLAB expression

**Syntax**
eval(expression)
[a1,a2,a3,...] = eval(function(b1,b2,b3,...))

**Description**
eval(expression) executes expression, a string containing any valid MATLAB expression. You can construct expression by concatenating substrings and variables inside square brackets:

expression = [string1,int2str(var),string2,...]

[a1,a2,a3,...] = eval(function(b1,b2,b3,...)) executes function with arguments b1,b2,b3,..., and returns the results in the specified output variables.

**Remarks**
Using the eval output argument list is recommended over including the output arguments in the expression string. The first syntax below avoids strict checking by the MATLAB parser and can produce untrapped errors and other unexpected behavior.

eval('[a1,a2,a3,...] = function(var)') % not recommended
[a1,a2,a3,...] = eval('function(var)') % recommended syntax

**Examples**
This for loop generates a sequence of 12 matrices named M1 through M12:

```matlab
for n = 1:12
    magic_str = ['M',int2str(n),', = magic(n)';
    eval(magic_str)
end
```

The next example executes the size function on a 3-dimensional array, returning the array dimensions in output variables d1, d2, and d3.

```matlab
A = magic(4);
A(:,:,2) = A';
[d1,d2,d3] = eval('size(A)')
```
d1 =
   4

d2 =
   4

d3 =
   2

See Also  assignin, catch, evalin, feval, lasterr, try
evalc

**Purpose**
Evaluate MATLAB expression with capture

**Syntax**

\[ T = \text{evalc}(S) \]
\[ T = \text{evalc}(s1,s2) \]
\[ [T,X,Y,Z,...] = \text{evalc}(S) \]

**Description**

\( T = \text{evalc}(S) \) is the same as \( \text{eval}(S) \) except that anything that would normally be written to the command window is captured and returned in the character array \( T \) (lines in \( T \) are separated by \( \backslash n \) characters).

\( T = \text{evalc}(s1,s2) \) is the same as \( \text{eval}(s1,s2) \) except that any output is captured into \( T \).

\[ [T,X,Y,Z,...] = \text{evalc}(S) \] is the same as \[ X,Y,Z,... = \text{eval}(S) \] except that any output is captured into \( T \).

**Remark**
When you are using evalc, diary, more, and input are disabled.

**See Also**
diary, eval, evalin, input, more
**Purpose**

Execute a string containing a MATLAB expression in a workspace

**Syntax**

```
evalin(ws,expression)
[a1,a2,a3,...] = evalin(ws,expression)
evalin(ws,expression,catch_expr)
```

**Description**

`evalin(ws,expression)` executes `expression`, a string containing any valid MATLAB expression, in the context of the workspace `ws`. `ws` can have a value of `base` or `caller` to denote the MATLAB base workspace or the workspace of the caller function. You can construct `expression` by concatenating substrings and variables inside square brackets:

```
expression = [string1,int2str(var),string2,...]
```

```
[a1,a2,a3,...] = evalin(ws,expression)
```

executes `expression` and returns the results in the specified output variables. Using the `evalin` output argument list is recommended over including the output arguments in the `expression` string:

```
evalin(ws,'[a1,a2,a3,...] = function(var)')
```

The above syntax avoids strict checking by the MATLAB parser and can produce untrapped errors and other unexpected behavior.

```
evalin(ws,expression,catch_expr)
```

executes `expression` and, if an error is detected, executes the `catch_expr` string. If `expression` produces an error, the error string can be obtained with the `lasterr` function. This syntax is useful when `expression` is a string that must be constructed from substrings. If this is not the case, use the `try...catch` control flow statement in your code.

**Remarks**

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the M-file. Note, the base and caller workspaces are equivalent in the context of an M-file that is invoked from the MATLAB command line.

**Examples**

This example extracts the value of the variable `var` in the MATLAB base workspace and captures the value in the local variable `v`:

```
v = evalin('base','var');
```
evalin cannot be used recursively to evaluate an expression. For example, a sequence of the form evalin('caller', evalin('caller', 'x')) doesn't work.

See Also assignin, catch, eval, feval, lasterr, try
Purpose

Check if variables or functions are defined

Graphical Interface

As an alternative to the exist function, use the Workspace browser or the Current Directory Browser.

Syntax

```matlab
exist item
exist item kind
a = exist('item','kind')
```

Description

exist('item') returns the status of item:

- 0 If item does not exist.
- 1 If item is a variable in the workspace.
- 2 If item is an M-file on your MATLAB search path. It also returns 2 when item is the full pathname to a file or when item is the name of an ordinary file on your MATLAB search path.
- 3 If item is a MEX- or DLL-file on your MATLAB search path.
- 4 If item is an MDL-file on your MATLAB search path.
- 5 If item is a built-in MATLAB function.
- 6 If item is a P-file on your MATLAB search path.
- 7 If item is a directory.
- 8 If item is a Java class.

If item specifies a filename, that filename may include an extension to preclude conflicting with other similar filenames. For example, exist('file.ext').

If item specifies a filename, MATLAB attempts to locate the file, examines the filename extension, and determines the value to return based on the extension alone. MATLAB does not examine the contents or internal structure of the file.

MEX, MDL, and P-files must be on the MATLAB search path for exist to return the values shown above. If item is found, but is not on the MATLAB search path, exist('item') returns 2, because it considers item to be an unknown file type.
exist

Any other file type or directory specified by item is not required to be on the MATLAB search path to be recognized by exist. If the file or directory is not on the search path, then item must specify either a full pathname, a partial pathname relative to MATLABPATH, or a partial pathname relative to your current directory.

If item is a Java class, then exist('item') returns an 8. However, if item is a Java class file, then exist('item') returns a 2.

eexist item kind returns the status of item for the specified kind. If item of type kind does not exist, it returns 0. The kind argument may be one of the following:

- builtin: Checks only for built-in functions.
- class: Checks only for Java classes.
- dir: Checks only for directories.
- file: Checks only for files or directories.
- var: Checks only for variables.

a = exist('item','kind') is the function form of the syntax.

Remarks
To check for the existence of more than one variable, use the ismember function. For example,

```matlab
a = 5.83;
c = 'teststring';
ismember({'a','b','c'},who)
an
```

Examples
This example uses exist to check whether a MATLAB function is a built-in function or a file:

```matlab
type = exist('plot')
type =
```

2-702
This indicates that `plot` is a built-in function.

In the following example, `exist` returns 8 on the Java class, `Welcome`, and returns 2 on the Java class file, `Welcome.class`.

```matlab
exist Welcome
ans =
    8

exist javaclasses/Welcome.class
ans =
    2
```

indicates there is a Java class `Welcome` and a Java class file `Welcome.class`.

The following example indicates that `testresults` is both a variable in the workspace and a directory on the search path:

```matlab
exist('testresults','var')
an =
    1

exist('testresults','dir')
an =
    7
```

### See Also

`assignin`, `computer`, `dir`, `evalin`, `help`, `inmem`, `isempty`, `lookfor`, `mfilename`, `partialpath`, `what`, `which`, `who`
**Purpose**
Terminate MATLAB (same as quit)

**Graphical Interface**
As an alternative to the exit function, select Exit MATLAB from the File menu or click the close box in the MATLAB desktop.

**Syntax**
exit

**Description**
exit ends the current MATLAB session. It is the same as quit, and takes the same termination options, such as force. For more information, see quit.

**See Also**
finish, quit
**Purpose**
Exponential

**Syntax**
\[ Y = \exp(X) \]

**Description**
The \( \exp \) function is an elementary function that operates element-wise on arrays. Its domain includes complex numbers.

\[ Y = \exp(X) \] returns the exponential for each element of \( X \). For complex \( z = x + iy \), it returns the complex exponential \( e^z = e^x(\cos(y) + i\sin(y)) \).

**Remark**
Use \( \expm \) for matrix exponentials.

**See Also**
\( \expm, \log, \log10, \expint \)
Purpose
Exponential integral

Syntax
Y = expint(X)

Definitions
The exponential integral computed by this function is defined as

\[ E_1(x) = \int_x^\infty \frac{e^{-t}}{t} dt \]

Another common definition of the exponential integral function is the Cauchy principal value integral

\[ Ei(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt \]

which, for real positive x, is related to expint as

\[ E_1(-x) = -Ei(x) - i\pi \]

Description
Y = expint(X) evaluates the exponential integral for each element of X.

References
Purpose
Matrix exponential

Syntax
Y = expm(X)

Description
Y = expm(X) raises the constant $e$ to the matrix power $X$. The expm function produces complex results if $X$ has nonpositive eigenvalues.

Use exp for the element-by-element exponential.

Algorithm
expm is a built-in function that uses the Padé approximation with scaling and squaring. You can see the coding of this algorithm in the expm1 demo.

Note
The expmdemo1, expmdemo2, and expmdemo3 demos illustrate the use of Padé approximation, Taylor series approximation, and eigenvalues and eigenvectors, respectively, to compute the matrix exponential.

References [1] and [2] describe and compare many algorithms for computing a matrix exponential. The built-in method, expm, is essentially method 3 of [2].

Examples
This example computes and compares the matrix exponential of $A$ and the exponential of $A$.

A = [1 1 0;
     0 0 2;
     0 0 -1 ];

expm(A)
ans =
  2.7183  1.7183  1.0862
     0  1.0000  1.2642
     0     0  0.3679

exp(A)
ans =
  2.7183  2.7183  1.0000
  1.0000  1.0000  7.3891
  1.0000  1.0000  0.3679
Notice that the diagonal elements of the two results are equal. This would be true for any triangular matrix. But the off-diagonal elements, including those below the diagonal, are different.

**See Also**

exp, funm, logm, sqrtm

**References**


Purpose
Compute $\exp(x) - 1$ accurately for small values of $x$

Syntax
$y = \text{expm1}(x)$

Description
$y = \text{expm1}(x)$ computes $\exp(x) - 1$, compensating for the roundoff in $\exp(x)$.
For small $x$, $\text{expm1}(x)$ is approximately $x$, whereas $\exp(x) - 1$ can be zero.

See Also
$\exp$, $\log1p$, $\text{expmdemo1}$
Purpose

Identity matrix

Syntax

\[
Y = \text{eye}(n)
\]
\[
Y = \text{eye}(m,n)
\]
\[
Y = \text{eye}(\text{size}(A))
\]
\[
\text{eye}(m, n, \text{classname})
\]
\[
\text{eye}([m,n], \text{classname})
\]

Description

\( Y = \text{eye}(n) \) returns the \( n \)-by-\( n \) identity matrix.

\( Y = \text{eye}(m,n) \) or \( \text{eye}([m \ n]) \) returns an \( m \)-by-\( n \) matrix with 1’s on the diagonal and 0’s elsewhere.

\( Y = \text{eye}(\text{size}(A)) \) returns an identity matrix the same size as \( A \).

\( \text{eye}(m, n, \text{classname}) \) or \( \text{eye}([m,n], \text{classname}) \) is an \( m \)-by-\( n \) matrix with 1’s of class \( \text{classname} \) on the diagonal and zeros of class \( \text{classname} \) elsewhere. \( \text{classname} \) is a string specifying the data type of the output. \( \text{classname} \) can have the following values: 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', or 'uint32'.

Example:

\[
x = \text{eye}(2,3,\text{'int8'})
\]

Limitations

The identity matrix is not defined for higher-dimensional arrays. The assignment \( y = \text{eye}([2,3,4]) \) results in an error.

See Also

ones, rand, randn, zeros
Purpose

Easy to use contour plotter

Syntax

ezcontour(f)

ezcontour(f,domain)

ezcontour(...,n)

ezcontour(axes_handle,...)

h = ezcontour(...)

Description

ezcontour(f) plots the contour lines of \( f(x,y) \), where \( f \) is a mathematical function of two variables, such as \( x \) and \( y \). ezcontour calls the contour function.

The function \( f \) is plotted over the default domain: \(-2\pi < x < 2\pi, -2\pi < y < 2\pi\). MATLAB chooses the computational grid according to the amount of variation that occurs; if the function \( f \) is not defined (singular) for points on the grid, then these points are not plotted.

\( f \) can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

\[
\text{ezcontour}(f, \text{domain}) \quad \text{plots } f(x,y) \text{ over the specified domain. domain can be either a 4-by-1 vector } \begin{bmatrix} x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}} \end{bmatrix} \text{ or a 2-by-1 vector } \begin{bmatrix} \text{min}, \text{max} \end{bmatrix} \text{ (where } \text{min} < x < \text{max}, \text{min} < y < \text{max}).
\]

If \( f \) is a function of the variables \( u \) and \( v \) (rather than \( x \) and \( y \)), then the domain endpoints \( u_{\text{min}}, u_{\text{max}}, v_{\text{min}}, \text{and } v_{\text{max}} \) are sorted alphabetically. Thus, \[
\text{ezcontour}('u^2 - v^3',[0,1],[3,6]) \quad \text{plots the contour lines for } u^2 - v^3 \text{ over } 0 < u < 1, 3 < v < 6.
\]

\[
\text{ezcontour}(... ,n) \quad \text{plots } f \text{ over the default domain using an } n \text{-by-} n \text{ grid. The default value for } n \text{ is 60.}
\]

\[
\text{ezcontour}(\text{axes_handle},...) \quad \text{plots into the axes with handle axes_handle instead of the current axes (gca).}
\]

\[
h = \text{ezcontour}(...) \quad \text{returns the handles to patch objects in } h.
\]

ezcontour automatically adds a title and axis labels.
**Remarks**

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezcontour`. For example, the MATLAB syntax for a contour plot of the expression

\[ \text{sqrt}(x^2 + y^2) \]

is written as

`ezcontour(‘sqrt(x^2 + y^2)’)`

That is, \( x^2 \) is interpreted as \( x.^2 \) in the string you pass to `ezcontour`.

**Passing a Function Handle**

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontour`.

\[
\begin{align*}
\text{fh} &= \ @(x,y) \ \text{sqrt}(x.^2 + y.^2); \\
\text{ezcontour}(\text{fh})
\end{align*}
\]

Note that when using function handles, you must use the array power, array multiplication, and array division operators (\(^.\), \(^.*\), \(^./\)) since `ezcontour` does not alter the syntax, as in the case with string inputs.

**Examples**

The following mathematical expression defines a function of two variables, \( x \) and \( y \).

\[
f(x,y) = 3(1-x)^2e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}
\]

`ezcontour` requires a function handle argument that expresses this function using MATLAB syntax. This example uses an anonymous function, which you can define in the command window without creating an M-File.

\[
f=\ @(x,y) \ 3*(1\cdot x)^2.*\exp(-\cdot(x.^2) - (y+1)^2) \ldots \\
-10\cdot(x/5 - x.^3 - y.^5).\exp(-x.^2\cdot y.^2) \ldots \\
-1/3\cdot\exp(-\cdot(x+1)^2 - y.^2); \]

For convenience, this function is written on three lines. See the peaks.

Pass the function handle `f` to `ezcontour` along with a domain ranging from \(-3\) to \(3\) in both \( x \) and \( y \) and specify a computational grid of 49-by-49:
In this particular case, the title is too long to fit at the top of the graph, so MATLAB abbreviates the string.

**See Also**

contour, ezcontourf, ezmesh, ezmeshc, ezplot, ezplot3, ezpolar, ezsurf, ezsurfc

“Contour Plots” for related functions
Purpose
Easy to use filled contour plotter

Syntax
- `ezcontourf(f)`
- `ezcontourf(f,domain)`
- `ezcontourf(...,n)`
- `ezcontourf(axes_handle,...)`
- `h = ezcontourf(...)`

Description
`ezcontourf(f)` plots the contour lines of \( f(x,y) \), where \( f \) is a string that represents a mathematical function of two variables, such as \( x \) and \( y \). `ezcontourf` calls the `contourf` function.

The function \( f \) is plotted over the default domain: \(-2\pi < x < 2\pi, -2\pi < y < 2\pi\). MATLAB chooses the computational grid according to the amount of variation that occurs; if the function \( f \) is not defined (singular) for points on the grid, then these points are not plotted.

\( f \) can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

`ezcontourf(f,domain)` plots \( f(x,y) \) over the specified domain. `domain` can be either a 4-by-1 vector \([x_{min}, x_{max}, y_{min}, y_{max}]\) or a 2-by-1 vector \([\text{min}, \text{max}]\) (where \( \text{min} < x < \text{max}, \text{min} < y < \text{max} \)).

If \( f \) is a function of the variables \( u \) and \( v \) (rather than \( x \) and \( y \)), then the domain endpoints \( u_{min}, u_{max}, v_{min}, \) and \( v_{max} \) are sorted alphabetically. Thus, `ezcontourf('u^2 - v^3',[0,1],[3,6])` plots the contour lines for \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

`ezcontourf(...,n)` plots \( f \) over the default domain using an \( n \)-by-\( n \) grid. The default value for \( n \) is 60.

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

`h = ezcontourf(...)` returns the handles to patch objects in \( h \).

`ezcontourf` automatically adds a title and axis labels.
Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to ezcontourf. For example, the MATLAB syntax for a filled contour plot of the expression

\[ \sqrt{x^2 + y^2}; \]

is written as

\[ \text{ezcontourf('}\sqrt{x^2 + y^2}'\text{)} \]

That is, \( x^2 \) is interpreted as \( x.^2 \) in the string you pass to ezcontourf.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle \( \text{fh} \) to ezcontourf.

\[
\text{fh} = @(x,y) \sqrt{x.^2 + y.^2};
\]

\[
\text{ezcontourf(fh)}
\]

Note that when using function handles, you must use the array power, array multiplication, and array division operators (\( .^\), \( .*\), \( ./\)) since ezcontourf does not alter the syntax, as in the case with string inputs.

Examples

The following mathematical expression defines a function of two variables, \( x \) and \( y \).

\[
f(x,y) = 3(1-x)^2e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5}-x^3-y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}
\]

ezcontourf requires a string argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the string

\[
f = ['3*(1-x)^2*exp(-(x^2)-(y+1)^2)','...
    '- 10*(x/5 - x^3 - y^5)*exp(-x^2-y^2)','...
    '- 1/3*exp(-(x+1)^2 - y^2')]\]

For convenience, this string is written on three lines and concatenated into one string using square brackets.
Pass the string variable \( f \) to `ezcontourf` along with a domain ranging from \(-3\) to \(3\) and specify a grid of 49-by-49:

\[
ezcontourf(f, [-3,3], 49)
\]

In this particular case, the title is too long to fit at the top of the graph, so MATLAB abbreviates the string.

**See Also**

- `contour`, `ezcontour`, `ezmesh`, `ezmeshc`, `ezplot`, `ezplot3`, `ezpolar`, `ezsurf`, `ezsurfc`

  “Contour Plots” for related functions
**Purpose**
Easy to use 3-D mesh plotter

**Syntax**
ezmesh(f)
ezmesh(f,domain)
ezmesh(x,y,z)
ezmesh(x,y,z,[smin,smax,tmin,tmax]) or ezmesh(x,y,z,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
ezmesh(axes_handle,...)
h = ezmesh(...)

**Description**
ezmesh(f) creates a graph of \( f(x,y) \), where \( f \) is a string that represents a mathematical function of two variables, such as \( x \) and \( y \). ezmesh calls the mesh function.

The function \( f \) is plotted over the default domain: \(-2\pi < x < 2\pi, -2\pi < y < 2\pi\). MATLAB chooses the computational grid according to the amount of variation that occurs; if the function \( f \) is not defined (singular) for points on the grid, then these points are not plotted.

\( f \) can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

ezmesh(f,domain) plots \( f \) over the specified domain. domain can be either a 4-by-1 vector \([x_{\text{min}},x_{\text{max}},y_{\text{min}},y_{\text{max}}]\) or a 2-by-1 vector \([\text{min},\text{max}]\) (where \( \text{min} < x < \text{max}, \text{min} < y < \text{max} \)).

If \( f \) is a function of the variables \( u \) and \( v \) (rather than \( x \) and \( y \)), then the domain endpoints \( u_{\text{min}}, u_{\text{max}}, v_{\text{min}}, \) and \( v_{\text{max}} \) are sorted alphabetically. Thus, ezmesh('u^2 - v^3',[0,1],[3,6]) plots \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

ezmesh(x,y,z) plots the parametric surface \( x = x(s,t), y = y(s,t), \) and \( z = z(s,t) \) over the square: \(-2\pi < s < 2\pi, -2\pi < t < 2\pi\).

ezmesh(x,y,z,[smin,smax,tmin,tmax]) or ezmesh(x,y,z,[min,max]) plots the parametric surface using the specified domain.

ezmesh(...,n) plots \( f \) over the default domain using an \( n \)-by-\( n \) grid. The default value for \( n \) is 60.
ezmesh(...,'circ') plots f over a disk centered on the domain.

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

h = ezmesh(...) returns the handles to a surface object in h.

Remarks

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to ezmesh. For example, the MATLAB syntax for a mesh plot of the expression

\[ \sqrt{x^2 + y^2}; \]

is written as

\[ \text{ezmesh('sqrt(x^2 + y^2)')} \]

That is, \(x^2\) is interpreted as \(x.^2\) in the string you pass to ezmesh.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle \(fh\) to ezmesh.

\[
fh = @(x,y) \sqrt{x^2 + y^2};
\]

\[ \text{ezmesh(fh)} \]

Note that when using function handles, you must use the array power, array multiplication, and array division operators (.^, .*, ./) since ezmesh does not alter the syntax, as in the case with string inputs.

Examples

This example visualizes the function

\[ f(x,y) = xe^{-x^2-y^2} \]

with a mesh plot drawn on a 40-by-40 grid. The mesh lines are set to a uniform blue color by setting the colormap to a single color:

\[
fh = @(x,y) x.*exp(-x.^2-y.^2);
\]

\[ \text{ezmesh(fh,40)} \]

\[ \text{colormap([0 0 1])} \]
See Also  

ezmeshc, mesh

"Function Plots" for related functions


**Purpose**
Easy to use combination mesh/contour plotter

**Syntax**
ezmeshc(f)
ezmeshc(f,domain)
ezmeshc(x,y,z)
ezmeshc(x,y,z,[smin,smax,tmin,tmax]) or ezmeshc(x,y,z,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
ezmeshc(axes_handle,...)
h = ezmeshc(...)

**Description**
ezmeshc(f) creates a graph of \( f(x,y) \), where \( f \) is a string that represents a mathematical function of two variables, such as \( x \) and \( y \). ezmeshc calls the meshc function.

The function \( f \) is plotted over the default domain \(-2\pi < x < 2\pi, -2\pi < y < 2\pi\). MATLAB chooses the computational grid according to the amount of variation that occurs; if the function \( f \) is not defined (singular) for points on the grid, then these points are not plotted.

\( f \) can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

ezmeshc(f,domain) plots \( f \) over the specified domain. domain can be either a 4-by-1 vector \([\text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}]\) or a 2-by-1 vector \([\text{min}, \text{max}]\) (where \( \text{min} < x < \text{max}, \text{min} < y < \text{max} \)).

If \( f \) is a function of the variables \( u \) and \( v \) (rather than \( x \) and \( y \)), then the domain endpoints \( \text{umin}, \text{umax}, \text{vmin}, \text{vmax} \) are sorted alphabetically. Thus, `ezmeshc('u^2 - v^3',[0,1],[3,6])` plots \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

ezmeshc(x,y,z) plots the parametric surface \( x = x(s,t), y = y(s,t), \text{and} \ z = z(s,t) \) over the square: \(-2\pi < s < 2\pi, -2\pi < t < 2\pi\).

ezmeshc(x,y,z,[smin,smax,tmin,tmax]) or ezmeshc(x,y,z,[min,max]) plots the parametric surface using the specified domain.

ezmeshc(...,n) plots \( f \) over the default domain using an \( n \)-by-\( n \) grid. The default value for \( n \) is 60.
ezmesh(. . , 'circ') plots $f$ over a disk centered on the domain.

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

$h = \text{ezmesh}(. . )$ returns the handles to a surface object in $h$.

**Remarks**

**Passing the Function as a String**

Array multiplication, division, and exponentiation are always implied in the expression you pass to ezmeshc. For example, the MATLAB syntax for a mesh/contour plot of the expression

$\sqrt{x^2 + y^2}$

is written as

```matlab
ezmeshc('\sqrt{(x^2 + y^2)}')
```

That is, $x^2$ is interpreted as $x.^2$ in the string you pass to ezmeshc.

**Passing a Function Handle**

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle $fh$ to ezmeshc.

```matlab
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmeshc(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators ($.^$, $.*$, $./$) since ezmeshc does not alter the syntax, as in the case with string inputs.

**Examples**

Create a mesh/contour graph of the expression

$$f(x, y) = \frac{y}{1 + x^2 + y^2}$$

over the domain $-5 < x < 5$, $-2\pi < y < 2\pi$:

```matlab
ezmeshc('y/(1 + x^2 + y^2)', [-5, 5, -2*pi, 2*pi])
```
Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65.5 and elevation = 26)

See Also ezmesh, ezsurfc, meshc

“Function Plots” for related functions
**Purpose**  
Easy to use function plotter

**Syntax**  
ezplot(f)  
ezplot(f,[min,max])  
ezplot(f,[xmin,xmax,ymin,ymax])  
ezplot(x,y)  
ezplot(x,y,[tmin,tmax])  
ezplot(...,figure_handle)  
eh = ezplot(...)  

**Description**  
ezplot(f) plots the expression $f = f(x)$ over the default domain $-2\pi < x < 2\pi$.  
f can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

ezplot(f,[min,max]) plots $f = f(x)$ over the domain: $\min < x < \max$.

For implicitly defined functions, $f = f(x,y)$:

ezplot(f) plots $f(x,y) = 0$ over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

ezplot(f,[xmin,xmax,ymin,ymax]) plots $f(x,y) = 0$ over $\text{min} < x < \text{xmax}$ and $\text{ymin} < y < \text{ymax}$.

ezplot(f,[min,max]) plots $f(x,y) = 0$ over $\min < x < \max$ and $\min < y < \max$.

If $f$ is a function of the variables $u$ and $v$ (rather than $x$ and $y$), then the domain endpoints $\text{umin}$, $\text{umax}$, $\text{vmin}$, and $\text{vmax}$ are sorted alphabetically. Thus,

ezplot('u^2 - v^2 - 1',[-3,2,-2,3]) plots $u^2 - v^2 - 1 = 0$ over $-3 < u < 2$, $-2 < v < 3$.

ezplot(x,y) plots the parametrically defined planar curve $x = x(t)$ and $y = y(t)$ over the default domain $0 < t < 2\pi$.

ezplot(x,y,[tmin,tmax]) plots $x = x(t)$ and $y = y(t)$ over $\text{tmin} < t < \text{tmax}$.

ezplot(...,figure_handle) plots the given function over the specified domain in the figure window identified by the handle figure.
ezplot(axes_handle,...) plots into the axes with handle axes_handle instead of the current axes (gca).

h = ezplot(...) returns the handles to a line objects in h.

Remarks

Passing the Function as a String
Array multiplication, division, and exponentiation are always implied in the expression you pass to ezplot. For example, the MATLAB syntax for a plot of the expression
\[ x^2 - y^2 \]
which represents an implicitly defined function, is written as

\[ \text{ezplot('x^2 - y^2')} \]

That is, \( x^2 \) is interpreted as \( x.^2 \) in the string you pass to ezplot.

Passing a Function Handle
Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle \( fh \) to ezplot.

\[
fh = @(x,y) \sqrt{x.^2 + y.^2};
\]
\[ \text{ezplot}(fh) \]

Note that when using function handles, you must use the array power, array multiplication, and array division operators (.^, .* , ./) since ezplot does not alter the syntax, as in the case with string inputs.

Examples
This example plots the implicitly defined function
\[ x^2 - y^4 = 0 \]
over the domain \([-2\pi, 2\pi]\):

\[ \text{ezplot('x^2-y^4')} \]
**See Also**

ezplot3, ezpolar, plot

“Function Plots” for related functions
ezplot3

Purpose
Easy to use 3-D parametric curve plotter

Syntax
\begin{align*}
\text{ezplot3}(x,y,z) \\
\text{ezplot3}(x,y,z,[t_{\text{min}},t_{\text{max}}]) \\
\text{ezplot3}(\ldots,'\text{animate}') \\
\text{ezplot3}(\text{axes}\_\text{handle},\ldots) \\
h = \text{ezplot3}(\ldots)
\end{align*}

Description
\text{ezplot3}(x,y,z) plots the spatial curve \( x = x(t), y = y(t), \) and \( z = z(t) \) over the default domain \( 0 < t < 2\pi \).

\( x, y, \) and \( z \) can be function handles for M-file functions or an anonymous functions (see Function Handles and Anonymous Functions) or strings (see the Remarks section).

\text{ezplot3}(x,y,z,[t_{\text{min}},t_{\text{max}}]) plots the curve \( x = x(t), y = y(t), \) and \( z = z(t) \) over the domain \( t_{\text{min}} < t < t_{\text{max}} \).

\text{ezplot3}(\ldots,'\text{animate}') produces an animated trace of the spatial curve.

\text{ezplot3}(\text{axes}\_\text{handle},\ldots) plots into the axes with handle \text{axes}\_\text{handle} instead of the current axes (\text{gca}).

\( h = \text{ezplot3}(\ldots) \) returns the handle to a line object in \( h \).

Remarks
Array multiplication, division, and exponentiation are always implied in the expression you pass to \text{ezplot3}. For example, the MATLAB syntax for a plot of the expression
\[ x = s./2, \ y = 2.*s, \ z = s.^2; \]
which represents a parametric function, is written as
\[ \text{ezplot3}('s/2','2*s','s^2') \]
That is, \( s/2 \) is interpreted as \( s/2 \) in the string you pass to \text{ezplot3}.

Passing a Function Handle
Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle \( \text{fh} \) to \text{ezplot3}.
fh1 = @(s) s./2; fh2 = @(s) 2.*s; fh3 = @(s) s.^2;
ezplot3(fh1,fh2,fh3)

Note that when using function handles, you must use the array power, array multiplication, and array division operators (.^, .* , ./) since ezplot does not alter the syntax, as in the case with string inputs.

**Examples**

This example plots the parametric curve

\[ x = \sin(t), \quad y = \cos(t), \quad z = t \]

over the domain \([0,6\pi]\):

\[ ezplot3(\text{‘sin(t)’,’cos(t)’,’t’,[0,6*pi]}) \]

See Also

ezplot, ezpolar, plot3

“Function Plots” for related functions
Purpose
ezpolar(f) plots the polar curve \( \rho = f(\theta) \) over the default domain \( 0 < \theta < 2\pi \).

\( f \) can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

ezpolar(f,[a,b]) plots \( f \) for \( a < \theta < b \).

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

\( h = \text{ezpolar}(...) \) returns the handles to a line object in \( h \).

Description
Array multiplication, division, and exponentiation are always implied in the expression you pass to ezpolar. For example, the MATLAB syntax for a plot of the expression
\[ t^2 \cdot \cos(t) \]
which represents an implicitly defined function, is written as
\[ \text{ezpolar('t^2*cos(t)')} \]
That is, \( t^2 \) is interpreted as \( t \cdot t \) in the string you pass to ezpolar.

Remarks
Passing a Function Handle
Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle \( \text{fh} \) to ezpolar.

\[ \text{fh} = @(t) t^2 \cdot \cos(t); \]
\[ \text{ezpolar(fh)} \]
Note that when using function handles, you must use the array power, array multiplication, and array division operators (.^,.*,./) since ezpolar does not alter the syntax, as in the case with string inputs.

**Examples**

This example creates a polar plot of the function

\[ r = 1 + \cos(t) \]

over the domain \([0, 2\pi]\):

```matlab
ezpolar('1+cos(t)')
```

See Also

ezplot, ezplot3, plot, plot3, polar

“Function Plots” for related functions
ezsurf

**Purpose**

Easy to use 3-D colored surface plotter

**Syntax**

- `ezsurf(f)`
- `ezsurf(f, domain)`
- `ezsurf(x, y, z)`
- `ezsurf(x, y, z, [smin, smax, tmin, tmax])` or `ezsurf(x, y, z, [min, max])`
- `ezsurf(..., n)`
- `ezsurf(..., 'circ')`
- `ezsurf(axess_handle, ...)`
- `h = ezsurf(...)`

**Description**

`ezsurf(f)` creates a graph of \( f(x,y) \), where \( f \) is a string that represents a mathematical function of two variables, such as \( x \) and \( y \). `ezsurf` calls the `surf` function.

The function \( f \) is plotted over the default domain: \(-2\pi < x < 2\pi, -2\pi < y < 2\pi\). MATLAB chooses the computational grid according to the amount of variation that occurs; if the function \( f \) is not defined (singular) for points on the grid, then these points are not plotted.

\( f \) can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

`ezsurf(f, domain)` plots \( f \) over the specified domain. domain can be either a 4-by-1 vector \([xmin, xmax, ymin, ymax]\) or a 2-by-1 vector \([min, max]\) (where \( min < x < max, min < y < max \)).

If \( f \) is a function of the variables \( u \) and \( v \) (rather than \( x \) and \( y \)), then the domain endpoints \( umin, umax, vmin, \) and \( vmax \) are sorted alphabetically. Thus, `ezsurf('u^2 - v^3', [0, 1], [3, 6])` plots \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

`ezsurf(x, y, z)` plots the parametric surface \( x = x(s,t), y = y(s,t), \) and \( z = z(s,t) \) over the square: \(-2\pi < s < 2\pi, -2\pi < t < 2\pi\).

`ezsurf(x, y, z, [smin, smax, tmin, tmax])` or `ezsurf(x, y, z, [min, max])` plots the parametric surface using the specified domain.

`ezsurf(..., n)` plots \( f \) over the default domain using an \( n \)-by-\( n \) grid. The default value for \( n \) is 60.
ezsurf(...) plots \( f \) over a disk centered on the domain.

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

\[ h = \text{ezsurf}(...) \] returns the handles to a surface object in \( h \).

**Remarks**

**Passing the Function as a String**

Array multiplication, division, and exponentiation are always implied in the expression you pass to ezmesh. For example, the MATLAB syntax for a surface plot of the expression

\[ \sqrt{x^2 + y^2}; \]

is written as

\[ \text{ezsurf('sqrt(x^2 + y^2)')} \]

That is, \( x^2 \) is interpreted as \( x.^2 \) in the string you pass to ezsurf.

**Passing a Function Handle**

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle \( fh \) to ezsurf.

\[ fh = @(x,y) \sqrt{x.^2 + y.^2}; \]
\[ \text{ezsurf}(fh) \]

Note that when using function handles, you must use the array power, array multiplication, and array division operators (\( .^, \times, \div \)) since ezsurf does not alter the syntax, as in the case with string inputs.

**Examples**

ezsurf does not graph points where the mathematical function is not defined (these data points are set to NaNs, which MATLAB does not plot). This example illustrates this filtering of singularities/discontinuous points by graphing the function

\[ f(x, y) = \text{real}(\tan(x + iy)) \]

over the default domain \(-2\pi < x < 2\pi, -2\pi < y < 2\pi:\)

\[ \text{ezsurf('real(atan(x+i*y))')} \]
Using \texttt{surf} to plot the same data produces a graph without filtering of discontinuities (as well as requiring more steps):

\begin{verbatim}
[x,y] = meshgrid(linspace(-2*pi,2*pi,60));
z = real(atan(x+i.*y));
surf(x,y,z)
\end{verbatim}
Note also that `ezsurf` creates graphs that have axis labels, a title, and extend to the axis limits.

**See Also**

- `ezmesh`, `ezsurfc`, `surf`
- “Function Plots” for related functions
ezsurfc

**Purpose**

Easy to use combination surface/contour plotter

**Syntax**

- `ezsurfc(f)`
- `ezsurfc(f, domain)`
- `ezsurfc(x, y, z)`
- `ezsurfc(x, y, z, [smin, smax, tmin, tmax])` or `ezsurfc(x, y, z, [min, max])`
- `ezsurfc(..., n)`
- `ezsurfc(..., 'circ')`
- `ezsurfc(axes_handle, ...)`
- `h = ezsurfc(...)`

**Description**

`ezsurfc(f)` creates a graph of \( f(x, y) \), where \( f \) is a string that represents a mathematical function of two variables, such as \( x \) and \( y \). `ezsurfc` calls the `surf` function.

The function \( f \) is plotted over the default domain: \(-2\pi < x < 2\pi, -2\pi < y < 2\pi\). MATLAB chooses the computational grid according to the amount of variation that occurs; if the function \( f \) is not defined (singular) for points on the grid, then these points are not plotted.

\( f \) can be a function handle for an M-file function or an anonymous function (see Function Handles and Anonymous Functions) or a string (see the Remarks section).

`ezsurfc(f, domain)` plots \( f \) over the specified domain. `domain` can be either a 4-by-1 vector \([\text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}]\) or a 2-by-1 vector \([\text{min}, \text{max}]\) (where \( \text{min} < x < \text{max} \), \( \text{min} < y < \text{max} \)).

If \( f \) is a function of the variables \( u \) and \( v \) (rather than \( x \) and \( y \)), then the domain endpoints \( \text{umin}, \text{umax}, \text{vmin}, \text{vmax} \) are sorted alphabetically. Thus, `ezsurfc('u^2 - v^3', [0, 1], [3, 6])` plots \( u^2 - v^3 \) over \( 0 < u < 1, 3 < v < 6 \).

`ezsurfc(x, y, z)` plots the parametric surface \( x = x(s, t), y = y(s, t), \) and \( z = z(s, t) \) over the square: \(-2\pi < s < 2\pi, -2\pi < t < 2\pi\).

`ezsurfc(x, y, z, [smin, smax, tmin, tmax])` or `ezsurfc(x, y, z, [min, max])` plots the parametric surface using the specified domain.

`ezsurfc(..., n)` plots \( f \) over the default domain using an \( n \)-by-\( n \) grid. The default value for \( n \) is 60.
ezsurfc(...) plots $f$ over a disk centered on the domain.

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

$h =$ ezsurfc(...) returns the handles to a surface object in $h$.

**Remarks**

**Passing the Function as a String**

Array multiplication, division, and exponentiation are always implied in the expression you pass to ezsurfc. For example, the MATLAB syntax for a surface/contour plot of the expression

$$\sqrt{x^2 + y^2};$$

is written as

$$\text{ezsurfc('sqrt(x^2 + y^2)')}$$

That is, $x^2$ is interpreted as $x .^2$ in the string you pass to ezsurfc.

**Passing a Function Handle**

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle $fh$ to ezsurfc.

```matlab
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators ($.^, .*, ./) since ezsurfc does not alter the syntax, as in the case with string inputs.

**Examples**

Create a surface/contour plot of the expression

$$f(x,y) = \frac{y}{1 + x^2 + y^2}$$

over the domain $-5 < x < 5$, $-2\pi < y < 2\pi$, with a computational grid of size 35-by-35:

```
ezsurf('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
```
Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65.5 and elevation = 26).

See Also

ezmesh, ezmeshc, ezsurf, surfc

“Function Plots” for related functions
Index

Symbols
! 2-17
- 2-2
% 2-17
& 2-13, 2-15
&& 2-15
' 2-2, 2-17
( ) 2-17
* 2-2
+ 2-2
, 2-17
. 2-17
... 2-17
/ 2-2
< 2-11
= 2-17
== 2-11
> 2-11
\ 2-2
^ 2-2
| 2-17
| 2-13, 2-15
|| 2-15
~ 2-13, 2-15
~= 2-11

A
abs 2-23
absolute value 2-23
accumarray 2-24
accuracy
  of linear equation solution 2-420
of matrix inversion 2-420
acos 2-28
acosh 2-30
acot 2-31
acoth 2-33
acsc 2-34
acsch 2-36
acscd 2-35
addpath 2-39
addressing selected array elements 2-21
addtodate 2-41, 2-252
adjacency graph 2-629
airy 2-42
Airy functions
  relationship to modified Bessel functions 2-42
ALim, Axes property 2-151
all 2-45
AmbientLightColor, Axes property 2-152
and (M-file function equivalent for &) 2-14
AND, logical
  bit-wise 2-239
angle 2-55
ans 2-81
any 2-82
arccosecant 2-34
arccosine 2-28
arccotangent 2-31
arcsecant 2-101
arcsine 2-106
arctangent 2-113
  four-quadrant 2-114, 2-115
arithmetic operations, matrix and array
distinguished 2-2
arithmetic operators
reference 2-2
array
addressing selected elements of 2-21
left division (arithmetic operator) 2-3
multiplication (arithmetic operator) 2-2
power (arithmetic operator) 2-3
right division (arithmetic operator) 2-3
shift circularly 2-353
transpose (arithmetic operator) 2-4
arrays
maximum size of 2-419
arrowhead matrix 2-410
ASCII
delimited files
writing 2-626
ASCII data
printable characters (list of) 2-334
reading 2-624
asec 2-101
asecd 2-103
asech 2-104
asin 2-106
asind 2-108
asinh 2-109
aspect ratio of axes 2-511
assignin 2-111
atan 2-113
atan2 2-114, 2-115
atand 2-117
atanh 2-118
.au files
reading 2-130
writing 2-131
audio
saving in AVI format 2-132
audioplayer 2-120
audiorecorder 2-124
aufinfo 2-129
auread 2-130
auwrite 2-131
avi 2-132
avifile 2-132
aviinfo 2-135
aviread 2-137
Axes
creating 2-138
defining default properties 2-142
fixed-width font 2-160
property descriptions 2-151
axes
setting and querying data aspect ratio 2-511
axis 2-174

B
BackgroundColor
annotation textbox property 2-76
balance 2-180
bar3 2-188
bar3h 2-188
BarLayout
barseries property 2-192
BarWidth
barseries property 2-192
base to decimal conversion 2-203
bar two operations
conversion from decimal to binary 2-569
Baseline
barseries property 2-192
BaseValue
area property 2-90
barseries property 2-192
beep 2-204
BeingDeleted
  areaseries property 2-90
  barseries property 2-193
  contour property 2-442
  errorbar property 2-679
Bessel functions
  first kind 2-212
  modified, first kind 2-209
  modified, second kind 2-215
  second kind 2-218
Bessel functions, modified
  relationship to Airy functions 2-42
Bessel's equation
  (defined) 2-212
  modified (defined) 2-209
besseli 2-209
besselj 2-212
besselnk 2-215
bessely 2-218
beta 2-221
beta function
  (defined) 2-221
  incomplete (defined) 2-223
  natural logarithm 2-224
betainc 2-223
betaln 2-224
bicg 2-225
bicgstab 2-232
BiConjugate Gradients method 2-225
BiConjugate Gradients Stabilized method 2-232
bin2dec 2-237
binary to decimal conversion 2-237
bitand 2-239
bitcmp 2-240
bitget 2-241
bitmax 2-242
bitor 2-244
bitset 2-245
bitshift 2-246
bit-wise operations
  AND 2-239
  get 2-241
  OR 2-244
  set bit 2-245
  shift 2-246
  XOR 2-248
bitxor 2-248
blanks
  removing trailing 2-566
blanks 2-249
blkdiag 2-250
box 2-251
Box, Axes property 2-152
braces, curly (special characters) 2-17
brackets (special characters) 2-17
break 2-252
breakpoints
  listing 2-543
  removing 2-533
  resuming execution from 2-535
  setting in M-files 2-546, 2-548
brighten 2-253
builtin 2-254
BusyAction
  areaseries property 2-90
  Axes property 2-152
  barseries property 2-193
  contour property 2-442
  errorbar property 2-679
ButtonDownFcn
  area series property 2-91
  Axes property 2-153
  barseries property 2-193
contour property 2-443
errorbar property 2-680
bvp4c 2-255
bvpget 2-264
bvpinit 2-265
bvpset 2-268

calendar 2-271
camdolly 2-272
camera
  dollying position 2-272
  moving camera and target positions 2-272
  placing a light at 2-276
  positioning to view objects 2-278
  rotating around camera target 2-280, 2-282
  rotating around viewing axis 2-286
  setting and querying position 2-283
  setting and querying projection type 2-285
  setting and querying target 2-287
  setting and querying up vector 2-289
  setting and querying view angle 2-291
CameraPosition, Axes property 2-153
CameraPositionMode, Axes property 2-153
CameraTarget, Axes property 2-153
CameraTargetMode, Axes property 2-154
CameraUpVector, Axes property 2-154
CameraUpVectorMode, Axes property 2-154
CameraViewAngle, Axes property 2-154
CameraViewAngleMode, Axes property 2-154
camlight 2-276
camlookat 2-278
camorbit 2-280
campan 2-282
campos 2-283
camproj 2-285
camroll 2-286
camtarget 2-287
camup 2-289
camva 2-291
camzoom 2-293
cart2pol 2-294
cart2sph 2-295
Cartesian coordinates 2-294, 2-295
case 2-296
cast 2-297
cat 2-298
catch 2-299
caxis 2-299

camera

cd 2-304
cd (ftp) 2-305
cdf2rdf 2-307
cdfepoch 2-309
cdfinfo 2-310
cdfread 2-313
cdfwrite 2-315

cell 2-318
cell 2-319
cell array
  creating 2-319
  structure of, displaying 2-328
  cell2mat 2-321
cell2struct 2-323
celldisp 2-325
cellfun 2-326
cellplot 2-328
cgs 2-330
char 2-334
checkin 2-336
  examples 2-337
  options 2-336
checkout 2-338
  examples 2-339
options 2-338
Children
areaseries property 2-91
Axes property 2-155
barseries property 2-194
contour property 2-443
errorbar property 2-680
chol 2-341
Cholesky factorization 2-341
    preordering for 2-410
cholinc 2-343
cholupdate 2-350
circshift 2-353
cla 2-354
clabel 2-355
class 2-361
clc 2-363, 2-369
clear 2-364
clearing
    Command Window 2-363
    items from workspace 2-364
    Java import list 2-365
clf 2-369
CLim, Axes property 2-156
CLimMode, Axes property 2-156
clipping 2-370
Clipping
    areaseries property 2-91
    Axes property 2-156
    barseries property 2-194
    contour property 2-443
    errorbar property 2-680
clock 2-371
close 2-372
    AVI files 2-374
    closest point search 2-641
cmapeditor 2-391
cmopts 2-377
colamd 2-378
colmmmd 2-380
colon operator 2-21
Color
    annotation arrow property 2-59
    annotation doublerarrow property 2-62
    annotation line property 2-67
    annotation textarrow property 2-71
    annotation textbox property 2-76
    Axes property 2-156
    errorbar property 2-680
colorbar 2-377
colormap
    editor 2-391
colormap 2-387
ColorOrder, Axes property 2-156
ColorSpec 2-408
colperm 2-410
comet 2-411
comet3 2-413
comma (special characters) 2-19
Command Window
    clearing 2-363
    get width 2-414
commandhistory 2-413
commandwindow 2-414
compan 2-415
companion matrix 2-415
compass 2-416
complementary error function
    (defined) 2-672
    scaled (defined) 2-672
complete elliptic integral
    (defined) 2-661
    modulus of 2-659, 2-661
complex
exponential (defined) 2-705
phase angle 2-55
complex 2-417
complex conjugate 2-428
    sorting pairs of 2-488
complex data
    creating 2-417
complex numbers, magnitude 2-23
computer 2-419
computer MATLAB is running on 2-419
concatenating arrays 2-298
cond 2-420
condeig 2-421
condest 2-422
condition number of matrix 2-420
    improving 2-180
coneplot 2-419
conj 2-428
conjugate, complex 2-428
    sorting pairs of 2-488
continuation ( . . . , special characters) 2-19
continue 2-429
contour
    and mesh plot 2-720
    filled plot 2-714
    functions 2-711
    of mathematical expression 2-712
    with surface plot 2-734
contour3 2-435
contourc 2-438
contourf 2-440
ContourMatrix
    contour property 2-443
contours
    in slice planes 2-456
contourslice 2-456
contrast 2-459
conv 2-460
conv2 2-461
conversion
    base to decimal 2-203
    binary to decimal 2-237
    Cartesian to cylindrical 2-294
    Cartesian to polar 2-294
    complex diagonal to real block diagonal 2-307
    decimal number to base 2-563, 2-568
    decimal to binary 2-569
    decimal to hexadecimal 2-570
    string matrix to cell array 2-329
    vector to character string 2-334
convex hulls
    multidimensional visualization 2-467
    two-dimensional visualization 2-465
convhull 2-465
convhulln 2-467
convn 2-469
cos 2-477
cosd 2-479
cosecant
    hyperbolic 2-496
    inverse 2-34
    inverse hyperbolic 2-36
cosh 2-480
cosine 2-477
  hyperbolic 2-480
  inverse 2-28
  inverse hyperbolic 2-30
cot 2-482
cotangent 2-482
  hyperbolic 2-485
  inverse 2-31
  inverse hyperbolic 2-33
cotd 2-484
coth 2-485
cov 2-487
cplxpair 2-488
cputime 2-489
CreateFcn
  areaseries property 2-91
  Axes property 2-157
  barseries property 2-194
  contour property 2-444
  errorbar property 2-680
cross 2-492
cross product 2-492
csc 2-493
cscd 2-495
csch 2-496
csvread 2-498
csvwrite 2-500
ctranspose (M-file function equivalent for ') 2-6
cumprod 2-501
cumsum 2-502
cumtrapz 2-503
cumulative
  product 2-501
  sum 2-502
curl 2-505
curly braces (special characters) 2-17
current directory

changing 2-304
CurrentPoint
  Axes property 2-157
customverctrl 2-508
cylinder 2-508
cylindrical coordinates 2-294, 2-295

D
daspect 2-511
data aspect ratio of axes 2-511
data types
  complex 2-417
DataAspectRatio, Axes property 2-158
DataAspectRatioMode, Axes property 2-159
datatipinfo 2-518
date 2-519
date and time functions 2-669
date string
  format of 2-523
date vector 2-531
datenum 2-520
datestr 2-523
datevec 2-531
dbclear 2-533
dbcont 2-535
dbdown 2-536
dblquad 2-537
dbmex 2-539
dbquit 2-540
dbstack 2-541
dbstatus 2-543
dbstep 2-545
dbstop 2-546
dbtype 2-553
dbup 2-554
dde23 2-555
Index

ddeget 2-559
ddeset 2-560
deal 2-563
debank 2-566
debugging
  changing workspace context 2-536
  changing workspace to calling M-file 2-554
  displaying function call stack 2-541
  MEX-files on UNIX 2-539
  quitting debug mode 2-540
  removing breakpoints 2-533
  resuming execution from breakpoint 2-545
  setting breakpoints in 2-546, 2-548
  stepping through lines 2-545

dec2base 2-563, 2-568
dec2bin 2-569
dec2hex 2-570
decic function 2-571
decimal number to base conversion 2-563, 2-568
decimal point (.) 2-18
  to distinguish matrix and array operations 2-2

decomposition
  Dulmage-Mendelsohn 2-629
dec2base 2-573
dec2bin 2-573
dec2hex 2-574
dec2hex 2-574
delaunay 2-577
  Delaunay tessellation
    3-dimensional visualization 2-582
    multidimensional visualization 2-585
  Delaunay triangulation
    visualization 2-577
delaunay3 2-582
delaunayn 2-585
delete 2-588
delete
timer object 2-591
delete (ftp) 2-590
DeleteFcn
  areaseries property 2-92
  Axes property 2-159
  barseries property 2-194
  contour property 2-444
  errorbar property 2-681
deleting
  files 2-588
  items from workspace 2-364
delimiters in ASCII files 2-624, 2-626
demo 2-592
depdir 2-596
depfun 2-597
derivative
  approximate 2-611
det 2-601
determinant of a matrix 2-601
detrend 2-602
deval 2-604
diag 2-606
diagonal 2-606
  main 2-606
dialog box
  error 2-691
diff 2-611
differences
  between adjacent array elements 2-611
differential equation solvers
  ODE boundary value problems 2-255
  adjusting parameters 2-268
  extracting properties 2-264
  extracting properties of 2-694, 2-695
  forming initial guess 2-265
dir 2-613
dir (ftp) 2-615
directories
  adding to search path 2-39
  checking existence of 2-701
  copying 2-470
  listing contents of 2-613
See also directory, search path
directory
  changing on FTP server 2-305
  listing for FTP server 2-615
See also directories
directory, changing 2-304
disconnect 2-375
discontinuities, plotting functions with 2-731
disp 2-617
disp
  timer object 2-618
display 2-620
DisplayName
  areaseeries property 2-92
  barseries property 2-195
  contour property 2-444
  errorbar property 2-681
distribution
  Gaussian 2-672
division
  array, left (arithmetic operator) 2-3
  array, right (arithmetic operator) 2-3
  matrix, left (arithmetic operator) 2-3
  matrix, right (arithmetic operator) 2-3
  of polynomials 2-573
dlmread 2-624
dlmwrite 2-626
dmperm 2-629
dolly camera 2-272
dos 2-634
dot 2-636
dot product 2-492, 2-636
double 2-637
double integral
  numerical evaluation 2-537
dragrect 2-638
DrawMode, Axes property 2-160
drawnow 2-639
dsearch 2-640
dsearchn 2-641
Dulmage-Mendelsohn decomposition 2-629
E
echo 2-642
disconnect 2-375
discontinuities, plotting functions with 2-731
disp 2-617
disp
  timer object 2-618
display 2-620
DisplayName
  areaseries property 2-92
  barseries property 2-195
  contour property 2-444
  errorbar property 2-681
distribution
  Gaussian 2-672
division
  array, left (arithmetic operator) 2-3
  array, right (arithmetic operator) 2-3
  matrix, left (arithmetic operator) 2-3
  matrix, right (arithmetic operator) 2-3
  of polynomials 2-573
dlmread 2-624
dlmwrite 2-626
dmperm 2-629
dolly camera 2-272
dos 2-634
dot 2-636
dot product 2-492, 2-636
double 2-637
double integral
  numerical evaluation 2-537
dragrect 2-638
DrawMode, Axes property 2-160
drawnow 2-639
dsearch 2-640
dsearchn 2-641
Dulmage-Mendelsohn decomposition 2-629
E
echo 2-642
discontinuities, plotting functions with 2-731
disp 2-617
disp
  timer object 2-618
display 2-620
DisplayName
  areaseries property 2-92
  barseries property 2-195
  contour property 2-444
  errorbar property 2-681
distribution
  Gaussian 2-672
division
  array, left (arithmetic operator) 2-3
  array, right (arithmetic operator) 2-3
  matrix, left (arithmetic operator) 2-3
  matrix, right (arithmetic operator) 2-3
  of polynomials 2-573
dlmread 2-624
dlmwrite 2-626
dmperm 2-629
dolly camera 2-272
dos 2-634
dot 2-636
dot product 2-492, 2-636
double 2-637
double integral
  numerical evaluation 2-537
dragrect 2-638
DrawMode, Axes property 2-160
drawnow 2-639
dsearch 2-640
dsearchn 2-641
Dulmage-Mendelsohn decomposition 2-629
E
echo 2-642
Index

eigenvector
  left 2-647
  right 2-647
eigs 2-651
ellipj 2-659
ellipke 2-661
ellipsoid 2-663
elliptic functions, Jacobian
  (defined) 2-659
elliptic integral
  complete (defined) 2-661
  modulus of 2-659, 2-661
else 2-664
elseif 2-665
end 2-667
end of line, indicating 2-19
eomday 2-669
eps 2-670
equal sign (special characters) 2-18
equations, linear
  accuracy of solution 2-420
EraseMode
  areaseries property 2-92
  barseries property 2-195
  contour property 2-444
  errorbar property 2-681
erf 2-672
erfc 2-672
erfcinv 2-672
erfcx 2-672
erfinv 2-672
error 2-674
error function
  (defined) 2-672
  complementary 2-672
  scaled complementary 2-672
error message
  displaying 2-674
  errorbars 2-676
  errordlg 2-691
etime 2-693
etree 2-694
etreeplot 2-695
eval 2-696
evalc 2-698
evalin 2-699
examples
  contouring mathematical expressions 2-712
  mesh plot of mathematical function 2-718
  mesh/contour plot 2-721
  plotting filled contours 2-715
  plotting function of two variables 2-724
  plotting parametric curves 2-727
  polar plot of function 2-729
  surface plot of mathematical function 2-731
  surface/contour plot 2-735
exclamation point (special characters) 2-19
execution
  resuming from breakpoint 2-535
exist 2-693
exit 2-704
exp 2-705
expint 2-706
expm 2-707
expm1 2-709
exponential 2-705
  complex (defined) 2-705
  integral 2-706
  matrix 2-707
exponentiation
  array (arithmetic operator) 2-3
  matrix (arithmetic operator) 2-3
eye 2-710
ezcontour 2-711
ezcontourf 2-714
ezmesh 2-717
ezmeshc 2-720
ezplot 2-723
ezplot3 2-726
ezpolar 2-728
ezsurf 2-730
ezsurfc 2-734
contour property 2-446
filter
two-dimensional 2-461
fixed-width font
axes 2-160
floating-point
integer, maximum 2-242
flow control
break 2-252
case 2-296
end 2-667
error 2-674
font
fixed-width, axes 2-160
FontAngle
annotation textbox property 2-78
Axes property 2-160
fontName
annotation textarrow property 2-71
annotation textbox property 2-78
Axes property 2-160
fonts
bold 2-72, 2-78
FontSize
annotation textarrow property 2-71
annotation textbox property 2-78
Axes property 2-161
FontUnits
Axes property 2-161
FontWeight
annotation textarrow property 2-72
annotation textbox property 2-78
Axes property 2-161
Fourier transform
convolution theorem and 2-460
functions
call stack for 2-541

F
FaceAlpha
annotation textbox property 2-76
FaceColor
annotation ellipse property 2-65
annotation rectangle property 2-69
areaseries property 2-93
barseries property 2-196
factorization, Cholesky 2-341
preordering for 2-410
Figures
updating from M-file 2-639
files
ASCII delimited
reading 2-624
writing 2-626
checking existence of 2-701
copying 2-470
deleting 2-588
deleting on FTP server 2-590
listing
names in a directory 2-613
size, determining 2-614
sound
reading 2-130
writing 2-131, 2-132
Fill
checking existence of 2-701
clearing from workspace 2-364

G
Gaussian distribution function 2-672
generalized eigenvalue problem 2-647
generating a sequence of matrix names (M1 through M12) 2-696
global variables, clearing from workspace 2-364

Graph
adjacency 2-629
graphics objects
Axes 2-138

graphics objects, deleting 2-588

GridLineStyle, Axes property 2-161

H
HandleVisibility
areaseries property 2-94
Axes property 2-161
barseries property 2-196
contour property 2-446
errorbar property 2-682

Head1Length
annotation doublearrow property 2-62

Head1Style
annotation doublearrow property 2-62

Head1Width
annotation doublearrow property 2-63

Head2Length
annotation doublearrow property 2-62

Head2Style
annotation doublearrow property 2-62

Head2Width
annotation doublearrow property 2-63

HeadLength
annotation arrow property 2-59
annotation textarrow property 2-72

HeadStyle
annotation arrow property 2-59
annotation textarrow property 2-72

HeadWidth
annotation arrow property 2-60

Height
annotation ellipse property 2-65
annotation rectangle property 2-69

Help browser
accessing from doc 2-630

HitTest
areaseries property 2-95
Axes property 2-162
barseries property 2-197
contour property 2-447
errorbar property 2-683

HitTestArea
areaseries property 2-95
barseries property 2-198
contour property 2-447
errorbar property 2-683

HorizontalAlignment
textarrow property 2-73
textbox property 2-78

horzcat (M-file function equivalent for [ , ] ) 2-19

Hyperbolic
cosecant 2-496
cosecant, inverse 2-36
cosine 2-480
cosine, inverse 2-30
cotangent 2-485
cotangent, inverse 2-33
secant, inverse 2-104
sine, inverse 2-109
tangent, inverse 2-118

I
identity matrix 2-710
incomplete beta function
(defined) 2-223
inheritance, of objects 2-362
integer
  floating-point, maximum 2-242
Interpreter
  textarrow property 2-73
  textbox property 2-79
Interruptible
  areaseries property 2-95
  Axes property 2-163
  barseries property 2-198, 2-199
  contour property 2-447
  errorbar property 2-684
inverse
  cosecant 2-34
  cosine 2-28
  cotangent 2-31
  hyperbolic cosecant 2-36
  hyperbolic cosine 2-30
  hyperbolic cotangent 2-33
  hyperbolic secant 2-104
  hyperbolic sine 2-109
  hyperbolic tangent 2-118
  secant 2-101
  sine 2-106
  tangent 2-113
  tangent, four-quadrant 2-114, 2-115
inversion, matrix
  accuracy of 2-420

J
Jacobian elliptic functions
(defined) 2-659
Java
  class names 2-365
  Java import list
    clearing 2-365
  joining arrays See concatenating arrays

L
labeling
  matrix columns 2-617
LabelSpacing
  contour property 2-448
Laplacian 2-574
Layer, Axes property 2-163
LData
  errorbar property 2-684
LDataSource
  errorbar property 2-684
ldivide (M-file function equivalent for \) 2-6
LevelList
  contour property 2-448
LevelListMode
  contour property 2-448
LevelStep
  contour property 2-448
LevelStepMode
  contour property 2-448
Light
  positioning in camera coordinates 2-276
line numbers in M-files 2-553
linear equation systems
  accuracy of solution 2-420
LineColor
  contour property 2-449
LineStyle
  annotation arrow property 2-60
  annotation doublearrow property 2-64
  annotation ellipse property 2-65
  annotation line property 2-67
  annotation rectangle property 2-69
  annotation textarrow property 2-73
  annotation textbox property 2-79
  areaseries property 2-96
  barseries property 2-198
  contour property 2-449
  errorbar property 2-685
LineStyleOrder
  Axes property 2-163
LineWidth
  annotation arrow property 2-60
  annotation doublearrow property 2-64
  annotation ellipse property 2-66
  annotation line property 2-67
  annotation rectangle property 2-70
  annotation textarrow property 2-73
  annotation textbox property 2-79
  areaseries property 2-96
  Axes property 2-164
  barseries property 2-199
  contour property 2-449
  errorbar property 2-685
Lobatto IIIa ODE solver 2-263
log
  saving session to file 2-609
logarithm
  of beta function (natural) 2-224
logical operations
  AND, bit-wise 2-239
  OR, bit-wise 2-244
  XOR, bit-wise 2-248
  logical operators 2-13, 2-15
logical tests
  all 2-45
  any 2-82
M
  Margin
    annotation textbox property 2-79
  Marker
    marker property 2-685
  MarkerFaceColor
    errorbar property 2-686
  MarkerEdgeColor
    errorbar property 2-686
  MarkerSize
    errorbar property 2-686
matrix
  addressing selected rows and columns of 2-21
  arrowhead 2-410
  companion 2-415
  condition number of 2-420
  condition number, improving 2-180
  converting to vector 2-21
  defective (defined) 2-648
  determinant of 2-601
  diagonal of 2-606
  Dulmage-Mendelsohn decomposition 2-629
  exponential 2-707
  identity 2-710
  inversion, accuracy of 2-420
  left division (arithmetic operator) 2-3
  maximum size of 2-419
  modal 2-646
  multiplication (defined) 2-2
  power (arithmetic operator) 2-3
  reading files into 2-624
  right division (arithmetic operator) 2-3
singularity, test for 2-601
trace of 2-606
transpose (arithmetic operator) 2-4
transposing 2-18
writing to ASCII delimited file 2-626
See also array
matrix names, (M1 through M12) generating a sequence of 2-696
matrix power See matrix, exponential
maximum matching 2-629
MDL-files
checking existence of 2-701
memory
    clearing 2-364
methods
    inheritance of 2-362
MEX-files
    clearing from workspace 2-364
    debugging on UNIX 2-539
M-file
    displaying during execution 2-642
    function file, echoing 2-642
    script file, echoing 2-642
M-files
    checking existence of 2-701
    clearing from workspace 2-364
    deleting 2-588
    editing 2-644
    line numbers, listing 2-553
    setting breakpoints 2-546, 2-548
MinorGridLineStyle, Axes property 2-164
minus (M-file function equivalent for -) 2-6
mldivide (M-file function equivalent for \) 2-6
mtimes (M-file function equivalent for *) 2-6
multidimensional arrays
    concatenating 2-298
multiplication
    array (arithmetic operator) 2-2
    matrix (defined) 2-2
    of polynomials 2-460
movies
    exporting in AVI format 2-132
mpower (M-file function equivalent for *) 2-6
mrdivide (M-file function equivalent for /) 2-6
mtimes (M-file function equivalent for *) 2-6
N
NextPlot
    Axes property 2-164
not (M-file function equivalent for ~) 2-14
numerical evaluation
    double integral 2-537
O
object
    inheritance 2-362
object classes, list of predefined 2-361
operating system command, issuing 2-19
operators
    arithmetic 2-2
    logical 2-13, 2-15
    overloading arithmetic 2-7
    overloading relational 2-11
    relational 2-11
    special characters 2-17
logical OR
    bit-wise 2-244
or (M-file function equivalent for |) 2-14
orthographic projection, setting and querying 2-285
OuterPosition
Axes property 2-165
overloading
arithmetic operators 2-7
relational operators 2-11
special characters 2-20

P
parametric curve, plotting 2-726
Parent
areareaseries property 2-96
Axes property 2-166
barseries property 2-199
contour property 2-449
errorbar property 2-687
parentheses (special characters) 2-18
path
adding directories to 2-39
pauses, removing 2-533
percent sign (special characters) 2-19
perfect matching 2-629
period (.), to distinguish matrix and array operations 2-2
period (special characters) 2-18
perspective projection, setting and querying 2-285
P-files
checking existence of 2-701
phase angle, complex 2-55
platform MATLAB is running on 2-419
PlotBoxAspectRatio, Axes property 2-166
PlotBoxAspectRatioMode, Axes property 2-166
plotting
contours (ez function) 2-711
ez-function mesh plot 2-717
filled contours 2-714
functions with discontinuities 2-731
in polar coordinates 2-728
mathematical function 2-723
mesh contour plot 2-720
parametric curve 2-726
surfaces 2-730
velocity vectors 2-423
plus (M-file function equivalent for +) 2-6
polar coordinates
computing the angle 2-55
converting from Cartesian 2-294
plotting in 2-728
polynomial
division 2-573
multiplication 2-460
poorly conditioned eigenvalues 2-180
Position
annotation textbox property 2-79
Axes property 2-167
position of camera
dollying 2-272
position of camera, setting and querying 2-283
power
matrix See matrix exponential
power (M-file function equivalent for .^) 2-6
printing, suppressing 2-19
product
cumulative 2-501
of vectors (cross) 2-492
scalar (dot) 2-492
projection type, setting and querying 2-285
ProjectionType, Axes property 2-167
R
rdivide (M-file function equivalent for ./) 2-6
rearrranging arrays
  converting to vector 2-21
rearranging matrices
  converting to vector 2-21
  transposing 2-18
reference page
  accessing from doc 2-630
regularly spaced vectors, creating 2-21
relational operators 2-11
rolling camera 2-286
rotating camera 2-280
rotating camera target 2-282
round
  towards infinity 2-318
roundoff error
  convolution theorem and 2-460
effect on eigenvalues 2-180
S
saving
  session to a file 2-609
scalar product (of vectors) 2-492
scaled complementary error function (defined) 2-672
search path
  adding directories to 2-39
secant
  inverse 2-101
  inverse hyperbolic 2-104
Selected
  areaseries property 2-96
  Axes property 2-167
  barseries property 2-199
  contour property 2-450
errorbar property 2-687
SelectionHighlight
  areaseries property 2-97
  Axes property 2-167
  barseries property 2-199
  contour property 2-450
  errorbar property 2-687
semicolon (special characters) 2-19
sequence of matrix names (M1 through M12)
  generating 2-696
session
  saving 2-609
shifting array
  circular 2-353
ShowText
  contour property 2-450
sine
  inverse 2-106
  inverse hyperbolic 2-109
single quote (special characters) 2-18
slice planes, contouring 2-456
sorting
  complex conjugate pairs 2-488
sound
  files
    reading 2-130
    writing 2-131
source control systems
  checking in files 2-336
  checking out files 2-338
  viewing current system 2-377
sparse matrix
  minimum degree ordering of 2-380
  permuting columns of 2-410
special characters
  overloading 2-20
spreadsheets
Index

reading into a matrix 2-624
writing matrices into 2-626
stack, displaying 2-541
str2cell 2-329
stretch-to-fill 2-139
String
textarrow property 2-73
textbox property 2-80
string
converting from vector to 2-334
string matrix to cell array conversion 2-329
subsref (M-file function equivalent for
A(i,j,k...)) 2-19
subtraction (arithmetic operator) 2-2
sum
cumulative 2-502
Surface
and contour plotter 2-734
plotting mathematical functions 2-730

T
Tag
areaseries property 2-97
Axes property 2-168
barseries property 2-199
contour property 2-450
errorbar property 2-687
tangent
four-quadrant, inverse 2-114, 2-115
inverse 2-113
inverse hyperbolic 2-118
target, of camera 2-287
test, logical See logical tests and detecting
TextColor
annotation textbarrow property 2-74
TextBackgroundColor
TextEdgeColor
TextLineWidth
TextList
contour property 2-450
TextListMode
contour property 2-451
TextMargin
annotation textbarrow property 2-74
TextRotation, textarrow property 2-74
TextStep
contour property 2-451
TextStepMode
contour property 2-451
TickDir, Axes property 2-168
TickDirMode, Axes property 2-168
TickLength, Axes property 2-168
time
CPU 2-489
required to execute commands 2-693
time and date functions 2-669
times (M-file function equivalent for .*) 2-6
Title, Axes property 2-169
trace of a matrix 2-606
trailing blanks
removing 2-566
transformation
See also conversion
transpose
array (arithmetic operator) 2-4
matrix (arithmetic operator) 2-4
transpose (M-file function equivalent for .') 2-6
truth tables (for logical operations) 2-13
Type
areaseries property 2-97
Axes property 2-169
barseries property 2-200
contour property 2-452
errorbar property 2-688

U
UData
errorbar property 2-688
UDataSource
errorbar property 2-688
UIContextMenu
areaseries property 2-97
Axes property 2-169
barseries property 2-200
contour property 2-452
errorbar property 2-688
uminus (M-file function equivalent for unary )
2-6

Units
Axes property 2-169
UNIX
Web browser 2-632
up vector, of camera 2-289
updating figure during M-file execution 2-639
uplus (M-file function equivalent for unary +) 2-6

UserData
areaseries property 2-97
Axes property 2-170
barseries property 2-200
contour property 2-452
errorbar property 2-688

V
variables
checking existence of 2-701

clearing from workspace 2-364
vector
dot product 2-636
product (cross) 2-492
vector field, plotting 2-423
vectorizing ODE function (BVP) 2-269
vectors, creating
regularly spaced 2-21
velocity vectors, plotting 2-423
vertcat (M-file function equivalent for [ ; ] ) 2-19
VerticalAlignment, textarrow property 2-74
VerticalAlignment, textbox property 2-80
video
saving in AVI format 2-132
view 2-278
view angle, of camera 2-291
View, Axes property (obsolete) 2-170
viewing
a group of object 2-278
a specific object in a scene 2-278
Visible
areaseries property 2-98
Axes property 2-170
barseries property 2-200
contour property 2-452
errorbar property 2-688
visualizing
cell array structure 2-328
volumes
contouring slice planes 2-456

W
Web browser
specifying for UNIX 2-632
Width
annotation ellipse property 2-66
annotation rectangle property 2-70
workspace
changing context while debugging 2-536, 2-554
  clearing items from 2-364

X
annotation arrow property 2-60, 2-64
annotation ellipse property 2-66
annotation line property 2-67
annotation rectangle property 2-70
annotation textarrow property 2-75
XAxisLocation, Axes property 2-170
XColor, Axes property 2-170
XData
  areaseris property 2-98
  barseries property 2-200
  contour property 2-452
  errorbar property 2-689
XDataMode
  areaseris property 2-98
  barseries property 2-201
  contour property 2-453
  errorbar property 2-689
XDataSource
  areaseris property 2-98
  barseries property 2-201
  contour property 2-453
  errorbar property 2-689
XDir, Axes property 2-171
XGrid, Axes property 2-171
XLabel, Axes property 2-171
XLim, Axes property 2-172
XLimMode, Axes property 2-172
XMinorGrid, Axes property 2-172
logical XOR
  bit-wise 2-248
XOR, printing 2-93, 2-196, 2-445, 2-682
XScale, Axes property 2-172
XTick, Axes property 2-172
XTickLabel, Axes property 2-173
XTickLabelMode, Axes property 2-173
XTickMode, Axes property 2-173

Y
annotation arrow property 2-61, 2-64, 2-68
annotation ellipse property 2-66
annotation rectangle property 2-70
annotation textarrow property 2-75
YAxisLocation, Axes property 2-170
YColor, Axes property 2-170
YData
  areaseris property 2-99
  barseries property 2-201
  contour property 2-453
  errorbar property 2-690
YDataMode
  contour property 2-453
YDataSource
  areaseris property 2-99
  barseries property 2-202
  contour property 2-454
  errorbar property 2-690
YDir, Axes property 2-171
YGrid, Axes property 2-171
YLabel, Axes property 2-171
YLim, Axes property 2-172
YLimMode, Axes property 2-172
YMinorGrid, Axes property 2-172
YScale, Axes property 2-172
YTick, Axes property 2-172
YTickLabel, Axes property 2-173
YTickLabelMode, Axes property 2-173
YTickMode, Axes property 2-173

Z
ZColor, Axes property 2-170
ZData
   contour property 2-454
ZDataSource
   contour property 2-454
ZDir, Axes property 2-171
ZGrid, Axes property 2-171
ZLim, Axes property 2-172
ZLimMode, Axes property 2-172
ZMinorGrid, Axes property 2-172
ZScale, Axes property 2-172
ZTick, Axes property 2-172
ZTickLabel, Axes property 2-173
ZTickLabelMode, Axes property 2-173
ZTickMode, Axes property 2-173